

Notas para final de PLP

Manuel Panichelli

December 17, 2023
v1.1

Contents

1	Paradigma funcional	4
1.1	Haskell	4
1.1.1	Programación funcional	4
1.1.2	Currificación	5
1.1.3	Pattern matching	6
1.1.4	Tipos recursivos	6
1.1.5	Listas	7
1.1.6	No terminación y orden de evaluación	7
1.1.7	Evaluación lazy	7
1.1.8	Esquemas de recursion	8
1.1.9	Transparencia referencial	9
1.1.10	Folds	9
1.2	Cálculo Lambda Tipado	13
1.2.1	Sistema de tipado	14
1.2.2	Resultados básicos	15
1.2.3	Semántica	16
1.3	Extensiones de Cálculo Lambda	20
1.3.1	λ^{bn} - Naturales	20
1.3.2	$\lambda^{\dots r}$ - Registros	22
1.3.3	λ^{bnu} - Unit	23
1.3.4	Referencias	24
1.3.5	Recursión	29
1.4	Inferencia de tipos	31
1.4.1	Definición formal de inferencia	31
1.4.2	Variables de tipos	32
1.4.3	Sustitución	32
1.4.4	Instancia de un juicio de tipado	33
1.4.5	Función de inferencia $\mathbb{W}(\cdot)$	33
1.4.6	Unificación	33
1.4.7	Algoritmo de inferencia	37
1.5	Subtipado	40
1.5.1	Principio de substitutividad	41
1.5.2	Reglas de subtipado	41
1.5.3	Subtipado en $\lambda^{\dots r}$	41
1.5.4	Subtipado de funciones	43
1.5.5	Subtipado de referencias	43
1.5.6	Algoritmo	45
2	Paradigma de objetos	47
2.1	Programación Orientada a Objetos	47
2.1.1	Method dispatch	47
2.1.2	Corrientes	48

2.1.3	Clasificación	48
2.1.4	Prototipado	51
2.2	Cálculo de objetos	51
2.2.1	Sintaxis	52
2.2.2	Atributos vs métodos	52
2.2.3	Variables libres	52
2.2.4	Sustitución	53
2.2.5	Equivalencia de términos (\equiv)	53
2.2.6	Semántica operacional	53
2.2.7	Ejemplo: Naturales	54
2.2.8	Codificando calculo λ	54
2.2.9	Codificación de clases	56
3	Paradigma lógico	59
3.1	Resolución para Lógica Proposicional	59
3.1.1	Lógica proposicional	59
3.1.2	Resolución	61
3.2	Lógica de Primer Orden	64
3.2.1	Sintaxis	64
3.2.2	Semántica	66
3.3	Resolución en lógica de primer orden	66
3.3.1	Forma clausal	67
3.3.2	Regla de resolución	70
3.3.3	Método de resolución	71
3.3.4	Diferencias con proposicional	72
3.4	Resolución SLD	73
3.4.1	Resolución lineal	73
3.4.2	Cláusulas de Horn	74
3.4.3	Resolución SLD	75
3.4.4	Refutación SLD	76
3.4.5	Corrección y completitud	77
3.5	Prolog	77
3.5.1	Resolución SLD en Prolog	77
3.5.2	Motivación de resolución SLD como cómputo	79
3.5.3	Notación Prolog	80
3.5.4	Ejemplos	80
3.5.5	Corrección y completitud	81
3.5.6	Refutaciones SLD en Prolog	81
3.5.7	Aspectos extra lógicos	82

Prefacio

Este es un apunte para el final de Paradigmas de Programación que preparé para rendir en la fecha de Julio 2022 (durante el cuatri). Tiene casi todo el contenido de las diapositivas de las clases de Diego del 1c2021, además de notas que fui agregando de las cosas que decía y aclaraciones que fui descubriendo mientras hacía ejercicios para el final.

Demás está decir que seguro tenga varios errores, que pueden ser tanto sintácticos como conceptuales. No tengan miedo de tirar PRs al repo [mnPanic/plp](#).

Control de cambios

v1.1 - 17/12/23

En 2c2023 estuve de ayudante en PLP, fui a las teóricas y corregí algunas cosas

- Haskell: algunas aclaraciones. Interpretación sintáctica alternativa de `foldr`. Nota sobre folds de estructuras algebraicas.
- Cálculo lambda: fixes en typos (¡gracias @facutotaro!). Algunas aclaraciones en semántica y tipado. Definición de forma normal.
- Inferencia: Fix ejemplo composición. Caso general algoritmo de unificación. Nota sobre ejecución en la práctica.
- Cálculo sigma: fix en clases, cambia λ por ς en constructor (¡gracias @VictoriaFiora!)

v1.0 - 15/6/22

- Versión inicial

Chapter 1

Paradigma funcional

1.1 Haskell

Def. 1.1 (Paradigma). Un **paradigma** es una forma de pensamiento.

Def. 1.2 (Lenguaje de programación). Un **lenguaje de programación** es el lenguaje que usamos para comunicar lo que queremos que haga una computadora.

Usamos un lenguaje para describir los computos que lleva a cabo la computadora.

Es **computacionalmente completo** si puede expresar todas las funciones computables. Hay DSLs (*domain specific languages*) que no pueden expresar todo lo computable.

Def. 1.3 (Paradigma de lenguaje de programación). Lo entendemos como un *estilo* de programación, que tiene que ver con los estilos de las soluciones. Está vinculado con lo que es para uno un modelo de cómputo.

Lo que vemos antes de la materia es el imperativo: a partir de un estado inicial llegar a un estado final. Programamos con secuencias de instrucciones para cambiar el estado.

1.1.1 Programación funcional

Definiciones:

- **Programa y modelo de cómputo:** Programar es definir funciones, y ejecutar es evaluar expresiones.
- **Programa:** Es un conjunto de ecuaciones. Por ej. `doble x = x + x`
- **Expresiones:** El significado de una expresión es su valor (si es que está definido). El valor de una expresión depende solo del valor de sus sub-expresiones. Evaluar o reducir una expresión es obtener su valor (por ej. `doble 2` \rightsquigarrow 4) No toda expresión denota un valor, por ejemplo `doble true`.
- **Tipos:** El universo de valores está particionado en colecciones denominadas *tipos*, que tienen operaciones asociadas.

Haskell es **fuertemente tipado**. Toda expresión bien formada tiene un tipo, que depende del tipo de sus subexpresiones. Si no puede asignarse un tipo a una expresión, no se la considera bien formada.

```
1           :: Int
'a'         :: Char
1.2        :: Float
True        :: Bool
```

```
[1, 2, 3]    :: [Int]
(1, True)   :: (Int, Bool)
succ        :: Int -> Int
```

Definiciones de funciones:

```
-- Definición
doble :: Int -> Int
doble x = x + x

-- Guardas
signo :: Int -> Bool
signo n | n >= 0    = True
        | otherwise = False

-- Definiciones locales
f (x, y) = g x + y
  where g z = z + 2

-- Expresiones lambda
\x -> x + 1
```

Tipos polimórficos

```
id x = x
id :: a -> a
-- x es de tipo a, que eventualmente se va a instanciar a algún tipo
```

Clases de tipos: Son como interfaces, que definen un conjunto de operaciones.

```
maximo :: Ord a => a -> a -> a
maximo x y | x > y = x
maximo _ y = y
-- Ord: (<), (<=), (>=), (>), max, min, compare
```

Tipos algebraicos

```
data Figura = Circulo Float | Rectangulo Float Float
deriving Eq -- deriva la igualdad nativa

-- (Circulo 1) == (Circulo 1)
```

Estas cosas nos permiten hacer funciones genéricas.

Funciones de alto orden: las funciones son first class citizens, se pueden pasar como parámetro.

1.1.2 Currificación

Es un mecanismo que permite reemplazar argumentos estructurados por una secuencia de argumentos "simples". Ventajas:

- Evaluación parcial: `succ = suma 1`
- Evita escribir paréntesis (asumiendo que la aplicación asocia a izquierda). `suma 1 2 = ((suma 1) 2)`

Por ejemplo, si definimos

```
add (x, y) = x + y
```

no está currificada, ya que recibe sus argumentos todos juntos. En cambio,

```
add x y = x + y
```

si está currificada.

Esto es porque escribir `add x y = x + y` es una conveniencia, pero lo que en realidad pasa por debajo es

```
add x y = x + y
add x = \y -> x + y
add = \x -> (\y -> x + y)
```

De esa forma, cada aplicación de un argumento devuelve una función, que podría ser aplicada de vuelta o no (y ahí tenemos aplicación parcial)

curry y uncurry

En criollo: una equivalencia entre una func con muchos parametros (una tupla) y una funcion equivalente que va tomando de a uno y devuelve funciones.

```
curry :: ((a, b) -> c) -> (a -> (b -> c))
curry f a b = f (a, b)

suma x y = x + y -- currificada
suma' :: (Int, Int) -> Int -- no currificada
suma' (x, y) = x + y

curry suma' 1 2 = suma' (1, 2)
curry suma' :: (Int -> (Int -> Int))

uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f (a, b) = f a b
-- es lo mismo que
-- uncurry f = (\(a, b) -> f a b)
```

1.1.3 Pattern matching

Una forma copada de definir funciones. Es un mecanismo para comparar un valor con un patrón. Si la comparación tiene éxito se puede deconstruir un valor en sus partes.

```
data Figura = Circulo Float | Rectangulo Float Float

area :: Figura -> Float
area (Circulo radio) = pi * radio ^ 2
area (Rectangulo l1 l2) = l1 * l2
```

El patrón está formado por el constructor y las variables. Los casos se evalúan en el orden en el que están escritos.

```
esCuadrado :: Figura -> Bool
-- También vale esto
-- esCuadrado (Rectangulo x y) = (x == y)
esCuadrado (Rectangulo x y) | (x==y) = True
esCuadrado _ = False
```

También se pueden definir funciones parciales (que no estén definidas para todo el dominio).

1.1.4 Tipos recursivos

La definición de un tipo puede tener uno o más parámetros del tipo

```
data Natural = Zero | Succ Natural
```

```

Zero :: Natural          -- 0
succ Zero :: Natural     -- 1
succ (succ (succ Zero)) :: Natural -- 2

dameNumero :: Natural -> Int
dameNumero Zero = 0
dameNumero (Succ n) = dameNumero n + 1

```

1.1.5 Listas

Tipo algebraico paramétrico recursivo con dos constructores:

```

[] :: [a]                -- lista vacia
(:) :: a -> [a] -> [a]  -- constructor infijo

-- Ejemplo
-- 1 : [2, 3] = [1, 2, 3]

```

Pattern matching

```

vacía :: [a] -> Bool
vacía [] = True
vacía _ = False

long :: [a] -> Int
long [] = 0
long x:xs = 1 + long xs

```

1.1.6 No terminación y orden de evaluación

```

-- No terminación
inf1 :: [Int]
inf1 = 1 : inf1

-- Evaluación no estricta
const :: a -> b -> a
const x y = x

-- const 42 inf1 -> 42 (pero depende del mecanismo de reducción del
-- lenguaje)

```

1.1.7 Evaluación lazy

el modelo de cómputo de haskell es la **reducción**. Se reemplaza un *redex* por otro usando las ecuaciones orientadas. Un redex (reducible expression) es una sub-expresión que no está en forma normal (irreducible).

Un redex debe ser una **instancia** del lado izquierdo de alguna ecuación y será reemplazado por el lado derecho con las variables correspondientes ligadas. El resto de la expresión no cambia.

Haskell hace esto hasta llegar a una forma normal, un valor irreducible.

`const x y = x`. `const x y` es un redex, y lo reduzco a `x`.

Y cómo selecciono una redex? **Orden normal** (lazy). Se selecciona el redex más externo para el que se pueda conocer que ecuación del programa utilizar. En general, primero las funciones más externas y luego los argumentos, solo de ser necesarios.

Modo aplicativo: reduce primero todos los argumentos. Se hace en otros lenguajes como c.

1.1.8 Esquemas de recursion

Formas de recursion comunes que uno puede aprovechar usando funciones de alto orden.

Map

```
-- tal que dobleL xs es la lista que contiene el doble de cada elemento en xs
dobleL :: [Float] -> [Float]
dobleL [] = []
dobleL (x:xs) = 2*x : dobleL xs

-- tal que la lista esParL xs indica si el correspondiente elemento en xs es par
-- o no
esParL :: [Int] -> [Bool]
esParL [] = []
esParL (x:xs) = (even x) : esParL xs

-- tal que longL xs es la lista que contiene las longitudes de las listas en xs
longL :: [[a]] -> [Int]
longL [] = []
longL (x:xs) = (length x) : longL xs

-- esquema recursivo de map:
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map g (x:xs) = g x : map g xs

-- Con eso, se pueden reescribir como
dobleL = map ((*) 2)
esParL = map even
longL = map length
```

Filter

```
-- tal que negativos xs contiene los elementos negativos de xs
negativos :: [Float] -> [Float]
negativos [] = []
negativos (x:xs)
  | x < 0 = x : (negativos xs)
  | otherwise = negativos xs

-- tal que la lista noVacías xs contiene las listas no vacías de xs
noVacías :: [[a]] -> [[a]]
noVacías [] = []
noVacías (l:ls)
  | (length l > 0) = l : (noVacías ls)
  | otherwise = noVacías ls

-- esquema recursivo:
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) = if (p x) then x : (filter p xs)
                  else (filter p xs)

-- luego quedan
```

```
negativos = filter (\x -> x < 0)
noVacías = filter (\l -> length l != 0)
noVacías = filter ((> 0) . length) --  $f \circ g = f(g(x))$ 
```

1.1.9 Transparencia referencial

El valor de una expresión en funcional depende solo de sus subexpresiones. Esto a diferencia de imperativo que depende del estado.

Si dos expresiones son iguales, denotan el mismo valor bajo el mismo contexto.

1.1.10 Folds

foldr

```
-- Funciones sobre listas
-- Motivación: Como estas funciones cambian la forma de lo que devolvemos, no lo
-- podemos hacer con map o filter.

-- sumaL: suma de todos los valores de una lista de enteros
sumaL :: [Int] -> Int
sumaL [] = 0
sumaL (x:xs) = x + (sumaL xs)

-- concat: la concatenación de todos los elementos de una lista de listas
concat :: [[a]] -> [a]
concat [] = []
concat (l:ls) = l ++ (concat ls)

-- reverso: el reverso de una lista
reverso :: [a] -> [a]
reverso [] = []
reverso (x:xs) = (reverso xs) ++ [x]

-- Esquema de recursión
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ z [] = z
foldr f z (x:xs) = f x (foldr f z xs)

-- Luego, con esto
sumaL = foldr (+) 0
concat = foldr (++) []
reverso = foldr (\x rec -> rec ++ [x]) []
reverso = foldr ( (flip (++) ) . (:[]) ) []

-- Hasta podemos definir map y filter. El fold es más general que el map y
-- filter
map f = foldr (\x rec -> f x : rec) []
map f = foldr ((:) . f) []

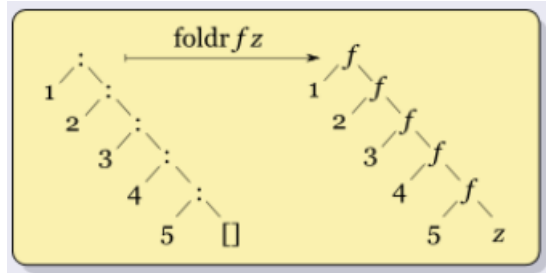
-- (:) . f :: a -> [b] -> [b]
-- ((:) . f) x = (f x) :

filter p = foldr (\x rec -> if p x then x : rec else rec) []
```

```
-- Longitud y suma con una sola pasada sobre la lista
sumaLong :: [Int] -> (Int, Int)
sumaLong = foldr (\x (r1, rn) -> (r1 + 1, rn + x)) (0, 0)
```

Definición de ++

```
(++) :: [a] -> [a] -> [a]
xs ++ ys = foldr (:) ys xs
```



Otra forma de pensarlo¹: reemplazar cada *cons* (:) por f, y el [] del final por z. *fold right* porque asume que el operador asocia a derecha.

```
foldr (+) 0
-- se puede ver como
1 : (2 : (3 : []))
1 + (2 + (3 + 0))
```

recr

```
-- dropWhile: elimina el prefijo más largo que cumpla con el predicado
-- la diferencia con recursión estructural es que tenemos disponible la cola
-- original
```

```
dropWhile :: (a -> Bool) -> [a] -> [a]
dropWhile _ [] = []
dropWhile p (x:xs) = if p x then dropWhile p xs else x:xs
```

```
-- ejemplo
dropWhile even [2, 4, 1, 6] = [1, 6]
```

```
-- No podemos hacerlo así nomás con fold porque no tenemos disponible la cola
-- sin recursión
-- \x r -> if p x then r else x:?
```

```
{-
drop while cuando se termina de cumplir devuelvo todo lo que viene "a la
derecha", pero cuando hago fold, lo que está a la derecha ya pasó por la
recursión.
Para poder hacerlo con foldr, nos guardamos la lista además de la recursión y
en cada paso decidimos si quedarnos con la recursión (cosas dropeadas) o la
lista original
-}
```

```
dropWhile' :: (a -> Bool) -> [a] -> ([a], [a])
dropWhile' p = foldr f ([], [])
  where f = (\x, (rs, os) -> if p x then rs else x: os)
```

¹Es la forma que usa el libro Programming In Haskell de Graham Hutton

```
dw p = first $ (foldr (\x (r1, r2)
    -> (if p x then r1 else x: r2, x: r2 ) ), ([], []))
```

*-- Nota: Una forma más fácil de pensar esto es primero hacer dropWhile' a mano,
-- con recursión explícita (sin ningún esquema) y luego mecánicamente pasarlo a fold*

Podemos otro esquema más poderoso: **recursión primitiva**. En cada paso, tengo acceso a la lista no modificada y la modificada. Este es el que usa dropWhile

```
-- esquema
g :: [a] -> b
g [] = z
g (x:xs) = f x xs (g xs)

{-
g [1, 2, 3]
= f 1 [2, 3] (g [2, 3])
= f 1 [2, 3] (f 2 [3] (g [3]))
= f 1 [2, 3] (f 2 [3] (f 3 [] (g [])))
= f 1 [2, 3] (f 2 [3] (f 3 [] z))
-}
```

```
recr :: b -> (a -> [a] -> b -> b) -> [a] -> b
recr z _ [] = z
recr z f (x:xs) = f x xs (recr z f xs)
```

-- Con esto podemos reescribir dropWhile sin hacer nada raro
dropWhile p = recr [] (\x xs rec -> if p x then rec else x:xs)

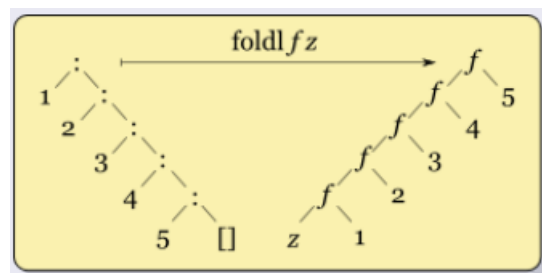
-- foldr en terminos de recr? Alcanza con ignorar xs
foldr f z = recr z (\x xs rec -> f x rec)

-- recr en términos de foldr?
recr z f = first \$
 foldr
 (\x (rs1, rs2) -> (f x rs2 rs1, x:rs2))
 (z, [])

```
recr z f = foldr f (z, [])  

    where f = (\x, (rec, xs) -> f x xs rec)
```

foldl



```
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl _ z [] = z
```

```
foldl f z (x:xs) = foldl f (f z x) xs
```

foldr = fold a la derecha, y foldl = fold a la izquierda. Se puede pensar como que asocia el operador a izquierda

```
foldl (#) v [x0, x1, ..., xn] = (... (((v # x0) # x1) # x2) ...) # xn
```

```
reverse = foldl (\c x -> x:c) []
```

```
reverse = foldl (flip (:)) []
```

Y uno en términos del otro? *Me falta repasar esto porque estaba matado, min 2:35:10 del video*

En listas infinitas, foldl se traba y foldr no. Esto es porque para terminar de evaluar, foldl tiene que llegar hasta el final de la lista. Pero si la función que uno aplica con foldr depende siempre de su segundo argumento (rec) para ser evaluada, también se va a trabar, por ejemplo con (+). Viendo el árbol queda más claro.

Fold sobre estructuras algebraicas

```
-- Arbol binario
```

```
data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)
```

```
-- Por ej.
```

```
Nodo 1 (Hoja 2) (Hoja 3)
```

```
-- Es
```

```
--
```

```
--   1
--  / \
-- 2   3
```

```
-- Y sobre ella podemos querer operaciones, como map
```

```
mapA :: (a -> b) -> Arbol a -> Arbol b
```

```
mapA f (Hoja x) = Hoja f x
```

```
mapA f (Nodo x izq der) = Nodo (f x) (mapA f izq) (mapA f der)
```

```
-- Y también podemos hacer un fold (esquema de recursión estructural)
```

```
foldA :: (a -> b) -> (a -> b -> b -> b) -> Arbol a -> b
```

```
foldA f g (Hoja x) = f x
```

```
foldA f g (Nodo x izq der) = g x (foldA f g izq) (foldA f g der)
```

```
sumaA = foldA id (\x rizq rder -> x + rizq + rder)
```

```
contarHojas = foldA 1 (\x rizq rder -> rizq + rder)
```

```
-- Por los tipos de los constructores, también podemos hacer la identidad
```

```
ida = foldrA (Hoja) (Nodo)
```

```
-- Arboles generales
```

```
data AG = NodoAG a [AG a]
```

```
mapAG :: (a -> b) -> AG a -> AG b
```

```
mapAG f (NodoAG a as) -> NodoAG (f a) (map (mapAG f) as)
```

```
foldAG :: (a -> [b] -> b) -> AG a -> b
foldAG f (NodoAG a as) = f a (map (foldAG f) as)
```

Aplicar un fold con un constructor es la identidad.

Práctica - Folds para tipos de datos algebraicos

Para definir un esquema de recursión estructural sobre un tipo de dato algebraico, en general si tenemos n constructores entonces vamos a recibir n funciones, una para cada uno (y con la misma aridad que el constructor). Para constructores que no toman argumentos (como podría ser Nil o []) se toma una constante.

Se puede generalizar la analogía de que se puede ver foldr como reemplazar sintácticamente cons por la función: reemplazás cada constructor por su función correspondiente. En el caso de listas, se reemplazaba cons por la función binaria y el constructor base (la lista vacía) por la constante.

1.2 Cálculo Lambda Tipado

Es el formalismo que está detrás de la programación funcional. Es un modelo de cómputo basado en **funciones**, introducido por Alonzo Church en 1934. Es computacionalmente completo en su versión sin tipos (turing completo) y nosotros vamos a estudiar la variante tipada (Church, 1941.)

La máquina de turing es más de estado, y este con reducción de expresiones.

Def. 1.4 (Tipos). Las **expresiones de tipos** (o tipos) de λ^b (lambda cálculo b de booleano) son

$$\sigma, \tau ::= Bool \mid \sigma \rightarrow \tau$$

Informalmente,

- $Bool$ es el tipo de los booleanos, y
- $\sigma \rightarrow \tau$ es el tipo de las funciones de tipo σ en tipo τ .

Ejemplo. Ejemplos:

- $Bool \rightarrow Bool$
- $Bool \rightarrow Bool \rightarrow Bool$

Def. 1.5 (Terminos). Los términos se escriben con las siguientes reglas de sintaxis.

Sea χ un conjunto infinito enumerable de variables, y $x \in \chi$. Los **términos** de λ^b están dados por,

$$\begin{aligned} M, N, P, Q &::= x \\ &\mid true \\ &\mid false \\ &\mid \text{if } M \text{ then } P \text{ else } Q \\ &\mid \lambda x : \sigma. M \\ &\mid M N \end{aligned}$$

Ejemplo. Ejemplos:

- $\lambda x : Bool. x$ ✓
- $\lambda x : Bool. \text{if } x \text{ then } false \text{ else } true$ ✓

- $\lambda f : Bool \rightarrow Bool \rightarrow Bool. \lambda x : Bool. f \ x \checkmark$
- $(\lambda f : Bool \rightarrow Bool. f \ true)(\lambda y : Bool. y) \checkmark$
- $true \ (\lambda x : Bool. x) \checkmark$
- $x \ y \checkmark$
- $\lambda x : true \ X$

1.2.1 Sistema de tipado

Es un sistema formal de deducción lógica o derivación que utiliza axiomas y reglas de inferencia para caracterizar un subconjunto de los términos llamados **tipados**. Nos permite quedarnos con algunos y rechazar otros términos en base a lo que consideremos correcto.

Definimos una **relación de tipado** en base a reglas de inferencia.

- Los **axiomas de tipado** establecen que ciertos **juicios de tipado** son derivables.
- Las **reglas de tipado** establecen que ciertos **juicios de tipado** son derivables siempre y cuando ciertos otros lo sean.

Def. 1.6 (Variables libres). Una variable puede ocurrir **libre** o ligada en un término. Decimos que x ocurre **libre** si no se encuentra bajo el alcance de una ocurrencia de λx . Caso contrario ocurre ligada.

Ejemplos:

- $\lambda x : Bool. \text{if } \underbrace{x}_{\text{ligada}} \text{ then } true \text{ else } false$
- $\lambda x : Bool. \lambda y : Bool. \text{if } true \text{ then } \underbrace{x}_{\text{ligada}} \text{ else } \underbrace{y}_{\text{ligada}}$
- $\lambda x : Bool. \text{if } \underbrace{x}_{\text{ligada}} \text{ then } true \text{ else } \underbrace{y}_{\text{libre}}$
- $(\lambda x : Bool. \text{if } \underbrace{x}_{\text{ligada}} \text{ then } true \text{ else } false) \underbrace{x}_{\text{libre}}$

La definición formal es a partir de cada término del lambda cálculo por pattern matching. $FV = \text{Free Variable}$

$$\begin{aligned}
 FV(x) &\stackrel{\text{def}}{=} \{x\} \\
 FV(true) = FV(false) &\stackrel{\text{def}}{=} \emptyset \\
 FV(\text{if } M \text{ then } P \text{ else } Q) &\stackrel{\text{def}}{=} FV(M) \cup FV(P) \cup FV(Q) \\
 FV(M \ N) &\stackrel{\text{def}}{=} FV(M) \cup FV(N) \\
 FV(\lambda x : \sigma. M) &\stackrel{\text{def}}{=} FV(M) \setminus \{x\}
 \end{aligned}$$

Def. 1.7 (Juicio de tipado). Un **juicio de tipado** es una expresión de la forma $\Gamma \triangleright M : \sigma$, que se lee “El término M tiene el tipo σ asumiendo el contexto de tipado Γ ”.

Un **contexto de tipado** es un conjunto de pares $x_i : \sigma_i$, notado $\{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$ donde los x_i son distintos. Usamos las letras Γ, Δ, \dots para contextos de tipado.

A las variables se les anota un tipo. Uno pone las asunciones que tiene sobre el tipo de algunas variables, como $x : Bool$.

Def. 1.8 (Axiomas de tipado de λ^b). Son guiadas por sintaxis al igual que las variables libres

$$\frac{}{\Gamma \triangleright true : Bool} (T-True) \quad \frac{}{\Gamma \triangleright false : Bool} (T-False)$$

(para cualquier contexto de tipado Γ)

$$\frac{x : \sigma \in \Gamma}{\Gamma \triangleright x : \sigma} (T-Var)$$

$$\frac{\Gamma \triangleright M : Bool \quad \Gamma \triangleright P : \sigma \quad \Gamma \triangleright Q : \sigma}{\Gamma \triangleright \text{if } M \text{ then } P \text{ else } Q : \sigma} (T-If)$$

P y Q tienen que tener el mismo tipo porque queremos que la expresion siempre tipe a lo mismo.

$$\frac{\Gamma, x : \sigma \triangleright M : \tau}{\Gamma \triangleright \lambda x : \sigma. M : \sigma \rightarrow \tau} (T-Abs) \quad \frac{\Gamma \triangleright M : \sigma \rightarrow \tau \quad \Gamma \triangleright N : \sigma}{\Gamma \triangleright M N : \tau} (T-App)$$

Si $\Gamma \triangleright M : \sigma$ puede derivarse usando los axiomas y reglas de tipado, decimos que es **derivable**, y decimos que M es **tipable** si el juicio de tipado $\Gamma \triangleright M : \sigma$ puede derivarse para algún Γ y σ .

Ejemplos:

- $\lambda x : Bool. x : Bool \rightarrow Bool$

$$\frac{\frac{\checkmark}{x : Bool \in \Gamma'}}{\Gamma' = \Gamma \cup \{x : Bool\} \triangleright x : Bool} (T-Var) \quad \frac{}{\Gamma \triangleright \lambda x : Bool. x : Bool \rightarrow Bool} (T-Abs)$$

- $\lambda x : Bool. \text{if } x \text{ then } false \text{ else } true$
- $\lambda f : Bool \rightarrow Bool \rightarrow Bool. \lambda x : Bool. f x$
- $(\lambda f : Bool \rightarrow Bool. f true)(\lambda y : Bool. y)$
- $true (\lambda x : Bool. x)$. No va a tipar nunca, porque $true : Bool$ y para $T - App$ necesitamos que sea $\sigma \rightarrow \tau$.
- $x y$. Para usar $T-App$, x por sintaxis solo aplica a $T-Var$. La única forma de que pueda aplicar x con y , x tiene que ser tipo flecha. Pero es una variable, entonces solo funcionaría si tenemos como asunción de tipo de x como función en Γ . Sin eso no se puede tipar.

1.2.2 Resultados básicos

Se pueden probar por inducción en la longitud de las reglas

Prop. 1.1 (Unicidad de tipos). Si $\Gamma \triangleright M : \sigma$ y $\Gamma \triangleright M : \tau$ son derivables, entonces $\sigma = \tau$.

Si una expresión tiene un tipo, ese tipo es único.

Prop. 1.2 (Weakening + Strengthening). Si $\Gamma \triangleright M : \sigma$ es derivable y $\Gamma \cap \Gamma'$ contiene a todas las variables libres de M , entonces $\Gamma' \triangleright M : \sigma$.

Puedo agrandar o achicar el contexto de tipo siempre y cuando contenga las mismas variables libres.

Otra: El tipo solo depende de las variables del contexto que aparecen libres en el término.

1.2.3 Semántica

Habiendo definido la sintaxis de λ^b , nos interesa formular como se **evalúan** o **ejecutan** los términos. Hay varias maneras de definir **rigurosamente** la semántica de un lenguaje de programación, nosotros vamos a definir una **semántica operacional**.

- Denotacional. Darle una *denotación* a cada símbolo del lenguaje, qué denota matemáticamente. Y uno define la semántica en términos de como las funciones van denotando cosas, con recursión o puntos fijos.
- Axiomática. Cuando cursamos algo1, definimos pre y pos condiciones. Predicados que definen el significado de una operación. Las triplas de hoare y esas cosas.
- **Operacional** consiste en
 - interpretar a los **términos como estados** de una máquina abstracta, y
 - definir una **función de transición** que indica dado un estado cuál es el siguiente.

El **significado** de un término M es el estado final que alcanza la máquina al comenzar con M como estado inicial. Hay dos formas de definir semántica operacional,

- **Small-step**: la función de transición describe un paso de computación. Esta vamos a hacer nosotros.
- **Big-step** (o **Natural Semantics**): la función de transición, en un paso, evalúa el término a su resultado.

Def. 1.9 (Juicios). La formulación se hace a través de **juicios de evaluación** $M \rightarrow N$, que se leen “el término M reduce, en un paso, al término N ”.

El significado de un juicio de evaluación se establece a través de:

- **Axiomas de evaluación**, que establecen que ciertos juicios de evaluación son derivables.
- **Reglas de evaluación**, que establecen que ciertos juicios de evaluación son derivables siempre y cuando ciertos otros lo sean

(análogo a axiomas y reglas de tipado)

Semántica operacional small-step de λ^b

Importante: A la hora de definir la semántica operacional, asumimos que ya pasamos por el proceso de tipado y que los términos tipan en el contexto vacío (i.e son términos cerrados, sin variables libres).

Además de introducir la función de transición es necesario introducir también los **valores**, los posibles resultados de evaluación de términos bien-tipados (derivables) y cerrados (sin variables libres).

Valores

$$V ::= true \mid false$$

todo término bien-tipado y cerrado de tipo Bool evalúa, en cero (directamente) o más pasos, a true o false. Se puede demostrar formalmente.

Juicio de evaluación en un paso:

$$\frac{}{\text{if } true \text{ then } M_2 \text{ else } M_3 \rightarrow M_2} \text{(E-IfTrue)}$$

$$\frac{}{\text{if } false \text{ then } M_2 \text{ else } M_3 \rightarrow M_3} \text{(E-IfFalse)}$$

$$\frac{M_1 \rightarrow M'_1}{\text{if } M_1 \text{ then } M_2 \text{ else } M_3 \rightarrow \text{if } M'_1 \text{ then } M_2 \text{ else } M_3} \text{(E-If)}$$

Esta última es una **regla de congruencia**: "progresa" la evaluación reduciendo un sub-término en un paso y permite reducir un término más grande.

Ejemplo de derivación

$$\begin{aligned} & \text{if } (\text{if } false \text{ then } false \text{ else } true) \text{ then } false \text{ else } true \\ & \rightarrow_{\text{(E-If, E-IfFalse)}} \text{if } true \text{ then } false \text{ else } true \\ & \rightarrow_{\text{(E-IfTrue)}} false \end{aligned}$$

Obs. 1.1. No existe M tal que $true \rightarrow M$ o $false \rightarrow M$. No los puedo reducir más.

Obs. 1.2. La estrategia de evaluación corresponde con el orden habitual de los lenguajes de programación.

1. Primero evaluar la guarda del condicional.
2. Una vez que la guarda sea un valor, seguir con la expresión del then o del else, según corresponda.

Por ejemplo,

$$\begin{aligned} & \text{if } true \text{ then } (\text{if } false \text{ then } false \text{ else } true) \text{ else } true \\ & \rightarrow \text{if } true \text{ then } true \text{ else } true \end{aligned}$$

y,

$$\begin{aligned} & \text{if } true \text{ then } (\text{if } false \text{ then } false \text{ else } true) \text{ else } true \\ & \rightarrow \text{if } false \text{ then } false \text{ else } true \end{aligned}$$

Lema 1.1 (Determinismo del juicio de evaluación en un paso). Si $M \rightarrow M'$ y $M \rightarrow M''$, entonces $M' = M''$.

Def. 1.10 (Forma normal). Una forma normal es un término que no puede reducirse o evaluarse más. i.e M tal que no existe N , $M \rightarrow N$.

Lema 1.2. Todo valor está en forma normal.

No vale el recíproco en λ^b , puedo tener cosas que están en forma normal pero que no sean valores, como términos que no estén bien tipados o que no sean cerrados. Ejemplos:

- $\text{if } x \text{ then } true \text{ else } false$: No tengo forma de reducir el x (es un término abierto).
- x . No tengo forma de reducirla pero no es ni $true$ ni $false$ (no es bool)
- $true \ false$ (no tipa).

Pero si vale en el cálculo de las expresiones booleanas cerradas.

Evaluación en muchos pasos

El juicio de **evaluación en muchos pasos** \rightarrow es la clausura reflexiva, transitiva de \rightarrow . Es decir, la menor relación tal que

1. Si $M \rightarrow M'$, entonces $M \rightarrow M'$
2. $M \rightarrow M$ para todo M .
3. Si $M \rightarrow M'$ y $M' \rightarrow M''$, entonces $M \rightarrow M''$.

captura la evolución en 0 y 1 pasos y la transitiva.

Ejemplo.

$\text{if } \text{true} \text{ then } (\text{if } \text{false} \text{ then } \text{false} \text{ else } \text{true}) \text{ else } \text{true} \rightarrow \text{true}$

Prop. 1.3 (Unicidad de formas normales). Si $M \rightarrow U$ y $M \rightarrow V$ con U, V formas normales, entonces $U = V$

aplicamos las reglas y llegamos a dos terminos entonces tienen que ser iguales.

Prop. 1.4 (Terminación). Para todo M existe una forma normal N tal que $M \rightarrow N$.

no me quedo ciclando, en una cantidad finita de pasos llego a una forma normal. N puede llegar a no ser un valor en algunos casos, eg: tiene variables libres (no es cerrada).

Extendiendo semántica operacional con funciones

Valores

$$V ::= \text{true} \mid \text{false} \mid \lambda x : \sigma. M$$

vamos a introducir una noción de evaluación tal que valgan los lemas previos y también el siguiente resultado

Teorema 1.1. Todo término bien tipado y cerrado de tipo

- *Bool* evalúa, en **cero o más** pasos a true o false.
- $\sigma \rightarrow \tau$ en **cero o más pasos** a $\lambda x : \sigma. M$, para alguna variable x y término M .

Juicios de evaluación en un paso (Además de E-IfTrue, E-IfFalse y E-If):

$$\frac{M_1 \rightarrow M'_1}{M_1 M_2 \rightarrow M'_1 M_2} (\text{E-App1} / \mu) \quad \text{primero reducís la función}$$

$$\frac{M_2 \rightarrow M'_2}{(\lambda x : \sigma. M) M_2 \rightarrow (\lambda x : \sigma. M) M'_2} (\text{E-App2} / \nu) \quad \text{luego reducís el "argumento"}$$

$$\frac{}{(\lambda x : \sigma. M) V \rightarrow M\{x \leftarrow V\}} (\text{E-AppAbs} / \beta) \quad \text{por último se hace la sustitución}$$

Las primeras dos son **reglas de congruencia** y la última de cómputo. Corresponde a una evaluación *call-by-value*, en donde se reduce el argumento y después se sustituye en el cuerpo de la función. La alternativa es *call-by-name*, que se sustituye tal cual y luego se reduce cuando se necesite (potencialmente más de una vez).

Sustitución

La operación,

$$M\{x \leftarrow N\}$$

quiere decir “Sustituir todas las ocurrencias **libres** de x en el término M por el término N ”. Es una operación importante que se usa para darle semántica a la aplicación de funciones. Es sencilla de definir pero requiere cuidado en el tratamiento de los ligadores de variables (λx).

Se define por sintaxis,

$$\begin{aligned} x\{x \leftarrow N\} &\stackrel{\text{def}}{=} N \\ a\{x \leftarrow N\} &\stackrel{\text{def}}{=} a \text{ si } a \in \{true, false\} \cup \chi \setminus \{x\} \\ (\text{if } M \text{ then } P \text{ else } Q)\{x \leftarrow N\} &\stackrel{\text{def}}{=} \text{if } M\{x \leftarrow N\} \text{ then } P\{x \leftarrow N\} \text{ else } N\{x \leftarrow N\} \\ (M_1 M_2)\{x \leftarrow N\} &\stackrel{\text{def}}{=} M_1\{x \leftarrow N\} M_2\{x \leftarrow N\} \\ (\lambda y : \sigma.M)\{x \leftarrow N\} &\stackrel{\text{def}}{=} \lambda y : \sigma.M\{x \leftarrow N\} \text{ si } x \neq y, y \notin \text{FV}(N) \end{aligned}$$

1. NB: La condición $x \neq y, y \notin \text{FV}(N)$ **siempre** puede cumplirse renombrando apropiadamente.
2. Técnicamente, la sustitución está definida sobre **clases de α -equivalencia de términos**.

α -equivalencia

Si en la siguiente expresión queremos sustituir la variable x por el término z ,

$$(\lambda z : \sigma.x)\{x \leftarrow z\} = \lambda z : \sigma.z$$

y lo hacemos de forma *naïve*, convertimos una función constante en la identidad. El problema es que $\lambda z : \sigma$ capturó la ocurrencia libre de z . Pero los nombres de las variables ligadas no son relevantes, la ecuación de arriba debería ser lo mismo que

$$(\lambda w : \sigma.x)\{x \leftarrow z\} = \lambda w : \sigma.z$$

para definir la sustitución sobre aplicaciones $(\lambda y : \sigma.M)\{x \leftarrow N\}$ vamos a asumir que la variable ligada y se renombró de forma tal que no ocurre libre en N .

Def. 1.11 (α -equivalencia). Dos términos M y N que difieren solamente en el nombre de sus variables ligadas se dicen α -equivalentes. Es una relación de equivalencia.

Ejemplo. Ejemplos:

- $\lambda x : Bool.x =_\alpha \lambda y : Bool.y$
- $\lambda x : Bool.y =_\alpha \lambda z : Bool.y$
- $\lambda x : Bool.y \neq_\alpha \lambda x : Bool.z$
- $\lambda x : Bool.\lambda x : Bool.x \neq_\alpha \lambda y : Bool.\lambda x : Bool.y$

La idea detrás es agrupar expresiones que sean semánticamente equivalentes.

Estado de error

Decimos que un término está en **forma normal** si no se puede seguir reduciendo.

$$M \text{ está en FN} \iff \nexists N. M \rightarrow N$$

Un **estado de error** es un término que no es un valor pero en el que la evaluación está trabada. (Un término en forma normal que no es un valor). Representa un estado en el cual el sistema de runtime en una implementación real generaría una excepción. Ejemplos:

- $\text{if } x \text{ then } M \text{ else } N$ (no es cerrado)
- $\text{true } M$ (no es tipable)

Objetivo de un sistema de tipos

El objetivo de un sistema de tipos es garantizar la **ausencia** de estados de error.

Def. 1.12. Decimos que un término **termina** o que es **fuertemente normalizante** si no hay cadenas de reducciones infinitas a partir de él.

Teorema 1.2. Todo término bien tipado termina. Si un término cerrado está bien tipado, entonces evalúa a un valor.

Esto es lo que nos gustaría que cumpla nuestro lenguaje.

Corrección

Decimos que **Corrección = Progreso + Preservación**.

Def. 1.13 (Progreso). Si M es cerrado y bien tipado, entonces

1. M es un valor, o bien
2. existe M' tal que $M \rightarrow M'$

La evaluación no puede trabarse para términos cerrados, bien tipados que no son valores.

Def. 1.14 (Preservación). Si $\Gamma \triangleright M : \sigma$ y $M \rightarrow N$, entonces $\Gamma \triangleright N : \sigma$.

La evaluación preserva tipos.

1.3 Extensiones de Cálculo Lambda

Cada vez que extendemos un lenguaje,

- Agregamos los **tipos** si hace falta,
- Extendemos los **términos**,
- Damos la **reglas de tipado**, y finalmente
- Damos la **semántica**

1.3.1 λ^{bn} - Naturales

Tipos y términos

$$\sigma ::= \text{Bool} \mid \text{Nat} \mid \sigma \rightarrow \rho$$

$$M ::= \dots \mid 0 \mid succ(M) \mid pred(M) \mid iszero(M)$$

Informalmente, la semántica de los términos es:

- $succ(M)$: Evaluar M hasta arrojar un número e incrementarlo.
- $pred(M)$: Evaluar M hasta arrojar un número y decrementarlo.
- $iszero(M)$: Evaluar M hasta arrojar un número, luego retornar $true$ | $false$ según sea cero o no.

agregamos términos para denotar ideas nuevas.

Axiomas y reglas de tipado:

$$\begin{array}{c} \overline{\Gamma \triangleright 0 : Nat} \text{ (T-Zero)} \\ \frac{\Gamma \triangleright M : Nat}{\Gamma \triangleright succ(M) : Nat} \text{ (T-Succ)} \quad \frac{\Gamma \triangleright M : Nat}{\Gamma \triangleright pred(M) : Nat} \text{ (T-Pred)} \\ \frac{\Gamma \triangleright M : Nat}{\Gamma \triangleright iszero(M) : Bool} \text{ (T-IsZero)} \end{array}$$

Valores

$$V ::= \dots \mid \underline{n} \quad \text{donde } \underline{n} \text{ abrevia } succ^n(0)$$

Juicio de evaluación en un paso

$$\begin{array}{c} \frac{M_1 \rightarrow M'_1}{succ(M_1) \rightarrow succ(M'_1)} \text{ (E-Succ)} \\ \frac{}{pred(0) \rightarrow 0} \text{ (E-PredZero)} \quad \frac{}{pred(\underline{n+1}) \rightarrow \underline{n}} \text{ (E-PredSucc)} \\ \frac{M_1 \rightarrow M'_1}{pred(M_1) \rightarrow pred(M'_1)} \text{ (E-Pred)} \\ \frac{}{iszero(0) \rightarrow true} \text{ (E-IsZeroZero)} \quad \frac{}{iszero(\underline{n+1}) \rightarrow false} \text{ (E-IsZeroSucc)} \\ \frac{M_1 \rightarrow M'_1}{iszero(M_1) \rightarrow iszero(M'_1)} \text{ (E-IsZero)} \end{array}$$

Además de las reglas de evaluación de un paso de λ^b (1.4 Tipos)

1.3.2 $\lambda^{\dots r}$ - Registros

Sea \mathcal{L} un conjunto de **etiquetas**, los tipos son:

$$\sigma ::= \dots \mid \{ l_i : \sigma_i^{i \in 1..n} \}$$

Ejemplos:

- $\{ \text{nombre} : \text{String}, \text{edad} : \text{Nat} \}$
- $\{ \text{persona} : \{ \text{nombre} : \text{String}, \text{edad} : \text{Nat} \}, \text{cuil} : \text{Nat} \}$
- Son posicionales, i.e

$$\{ \text{nombre} : \text{String}, \text{edad} : \text{Nat} \} \neq \{ \text{edad} : \text{Nat}, \text{nombre} : \text{String} \}$$

Términos:

$$M ::= \dots \mid \{ l_i = M_i^{i \in 1..n} \} \mid M.I$$

Informalmente, la semántica es

- El registro $\{ l_i = M_i^{i \in 1..n} \}$ evalúa a $\{ l_i = V_i^{i \in 1..n} \}$ con V_i el valor al que evalúa M_i .
- $M.I$ evalúa M hasta que sea un registro valor, luego proyecta el campo correspondiente.

Ejemplos,

- $\lambda x : \text{Nat}. \lambda y : \text{Bool}. \{ \text{edad} = x, \text{esMujer} = y \}$
- $\lambda p : \{ \text{edad} : \text{Nat}, \text{esMujer} : \text{Bool} \}. p.\text{edad}$
- $(\lambda p : \{ \text{edad} : \text{Nat}, \text{esMujer} : \text{Bool} \}. p.\text{edad}) \{ \text{edad} = 20, \text{esMujer} = \text{false} \}$

Tipado:

$$\frac{\Gamma \triangleright M_i : \sigma_i \text{ para cada } i \in 1..n}{\Gamma \triangleright \{ l_i = M_i^{i \in 1..n} \} : \{ l_i : \sigma_i^{i \in 1..n} \}} \text{(T-Rcd)}$$

$$\frac{\Gamma \triangleright M : \{ l_i : \sigma_i^{i \in 1..n} \} \quad j \in 1..n}{\Gamma \triangleright M.l_j : \sigma_j} \text{(T-Proj)}$$

Valores:

$$V ::= \dots \mid \{ l_i = V_i^{i \in 1..n} \}$$

Semántica operacional:

$$\begin{array}{c}
\frac{M_j \rightarrow M'_j}{\{ l_i = V_i \quad i \in 1..j-1, \quad l_j = M_j, \quad l_i = M_i \quad i \in j+1..n \}} \text{(E-Rcd)} \\
\rightarrow \\
\{ l_i = V_i \quad i \in 1..j-1, \quad l_j = M'_j, \quad l_i = M_i \quad i \in j+1..n \} \\
\text{(se reducen de izquierda a derecha)}
\end{array}$$

$$\begin{array}{c}
\frac{j \in 1..n}{\{ l_i = V_i \quad i \in 1..n \}. l_j \rightarrow V_j} \text{(E-ProjRcd)} \\
\frac{M \rightarrow M'}{M.I \rightarrow M'.I} \text{(E-Proj)}
\end{array}$$

El orden de reducción de izquierda a derecha es importante para **no perder el determinismo** en la semántica operacional.

1.3.3 λ^{bnu} - Unit

$$\begin{array}{l}
\sigma ::= Bool \mid Nat \mid \textcolor{blue}{Unit} \mid \sigma \rightarrow \rho \\
M ::= \dots \mid unit
\end{array}$$

Informalmente, *Unit* es un tipo unitario y el único valor posible de una expresión de ese tipo es *unit*. Es parecido a la idea de void en lenguajes como C o Java.

Tipado:

$$\frac{}{\Gamma \triangleright unit : Unit} \text{(T-Unit)}$$

Valores:

$$V ::= \dots \mid unit$$

No hay reglas de evaluación nuevas, ya que *unit* es un valor.

Su utilidad principal es en lenguajes con efectos laterales. Porque en ellos es útil poder evaluar varias expresiones en **secuencia**,

$$M_1; M_2 \stackrel{\text{def}}{=} (\lambda x : Unit. M_2) M_1 \quad x \notin \text{FV}(M_2)$$

- La evaluación de $M_1; M_2$ consiste en primero evaluar M_1 hasta que sea un valor V_1 , reemplazar las apariciones de x en M_2 por V_1 , y luego evaluar M_2 .
- Como no hay apariciones libres de x en M_2 , se evalúa M_1 y luego M_2 . Este comportamiento se logra con las reglas de evaluación definidas previamente.

1.3.4 Referencias

$\lambda^{...let}$ - **Ligado**

$$M ::= \dots \mid \text{let } x : \sigma = M \text{ in } N$$

Informalmente,

- $\text{let } x : \sigma = M \text{ in } N$ evaluar M a un valor V , ligar x a V y evaluar N .
- Mejora la legibilidad, y la extensión **no** implica agregar nuevos tipos, es solo sintaxis.

Ejemplos,

- $\text{let } x : \text{Nat} = \underline{2} \text{ in } \text{succ}(x) \rightsquigarrow \underline{3}$
- $\text{pred}((\text{let } x : \text{Nat} = \underline{2} \text{ in } x))$
- $\text{let } f : \text{Nat} \rightarrow \text{Nat} = \lambda x : \text{Nat}. \text{succ}(n) \text{ in } f(f \ 0)$

Como se puede ver, sirve para nombrar cosas que se usan más de una vez o dar declaratividad.

- $\text{let } x : \text{Nat} = \underline{2} \text{ in } \text{let } x : \text{Nat} = \underline{3} \text{ in } x$

Cual es la diferencia entre $\text{let } x : \text{Nat} = \underline{2} \text{ in } M$ y $(\lambda x : \text{Nat}. M) \ \underline{2}$? Sirve cuando lo querés usar más de una vez

Tipado,

$$\frac{\Gamma \triangleright M : \sigma_1 \quad \Gamma, x : \sigma_1 \triangleright N : \sigma_2}{\Gamma \triangleright \text{let } x : \sigma_1 = M \text{ in } N : \sigma_2} (\text{T-Let})$$

Semántica operacional,

$$\frac{M_1 \rightarrow M'_1}{\text{let } x : \sigma = M_1 \text{ in } M_2 \rightarrow \text{let } x : \sigma = M'_1 \text{ in } M_2} (\text{E-Let})$$

$$\frac{}{\text{let } x : \sigma = \mathbf{V}_1 \text{ in } M_2 \rightarrow M_2\{x \leftarrow \mathbf{V}_1\}} (\text{E-LetV})$$

*el objetivo es llevar a M_1 a un valor, y luego reemplazar las apariciones libres de x en M_2 por V_1 . Es parecido a la aplicación, y es como el *where* de haskell pero invertido*

Motivación

En una expresión como

$$\text{let } x : \text{Nat} = \underline{2} \text{ in } M,$$

x es una variable declarada con valor 2. Pero el valor de x permanece **inalterado** a lo largo de la evaluación de M . Es **immutable**, no existe una operación de asignación.

En programación imperativa pasa todo lo contrario, todas las variables son mutables. Por eso vamos a extender al cálculo lambda tipado con variables mutables.

Operaciones básicas

- **Alocación** (Reserva de memoria), $\text{ref } M$ genera una referencia fresca cuyo contenido es el valor de M .
- **Derreferenciación** (Lectura), $!x$ sigue la referencia x y retorna su contenido.
- **Asignación**. $x := M$ almacena en la referencia x el valor de M .

Ejemplos

sin tipos en las let-expresiones para facilitar la lectura

- $\text{let } x = \text{ref } \underline{2} \text{ in } !x$ evalúa a $\underline{2}$
- $\text{let } x = \text{ref } \underline{2} \text{ in } (\lambda_ : \text{unit}.!x) (x := \text{succ}(!x))$ evalúa a $\underline{3}$
- **Por qué no así?** $\text{let } x = \text{ref } \underline{2} \text{ in } x := \text{succ}(!x); !x$
- $\text{let } x = \underline{2} \text{ in } x$ evalúa a $\underline{2}$
- $\text{let } x = \text{ref } \underline{2} \text{ in } x$ evalúa a **ver**
- **aliasing**, $\text{let } x = \text{ref } \underline{2} \text{ in let } y = x \text{ in } (\lambda.\text{unit}.!x) (x := \text{succ}(!y))$

x e y son *alias* para la misma celda de memoria. $!x == !y$.

El término

$$\text{let } x = \text{ref } \underline{2} \text{ in } x := \text{succ}(!x)$$

Comandos

a qué evalúa? La asignación interesa por su **efecto**, y no su valor. No tiene interés preguntarse por el valor pero sí tiene sentido por el efecto.

Def. 1.15 (Comando). Vamos a definir un **Comando** como una expresión que se evalúa para causar un efecto, y a *unit* como su valor.

Un lenguaje funcional **puro** es uno en el que las expresiones son **puras** en el sentido de carecer de efectos. Este lenguaje ya no es funcional puro.

Tipos y términos

Las **expresiones de tipos** ahora son

$$\sigma ::= \text{Bool} \mid \text{Nat} \mid \sigma \rightarrow \tau \mid \text{Unit} \mid \text{Ref } \sigma$$

Informalmente, $\text{Ref } \sigma$ es el tipo de las referencias de valores de tipo sigma. Por ej. $\text{Ref } (\text{Bool} \rightarrow \text{Nat})$ es el tipo de las referencias a funciones de *Bool* en *Nat*.

Términos,

$$\begin{aligned}
M ::= & x \\
& | \lambda x : \sigma. M \\
& | M \ N \\
& | \text{unit} \\
& | \text{ref } M \\
& | !M \\
& | M := N \\
& | \dots
\end{aligned}$$

Reglas de tipado,

$$\frac{\Gamma \triangleright M_1 : \sigma}{\Gamma \triangleright \text{ref } M_1 : \text{Ref } \sigma} (\text{T-Ref})$$

$$\frac{\Gamma \triangleright M_1 : \text{Ref } \sigma}{\Gamma \triangleright !M_1 : \sigma} (\text{T-DeRef})$$

$$\frac{\Gamma \triangleright M_1 : \text{Ref } \sigma \quad \Gamma \triangleright M_2 : \sigma}{\Gamma \triangleright M_1 := M_2 : \text{Unit}} (\text{T-Assign})$$

Semántica operacional

Qué es una referencia? Es una abstracción de una porción de memoria que se encuentra en uso. Vamos a usar **direcciones** o **locations**, $l, l_i \in \mathcal{L}$ para representar referencias. Una **memoria** o **store** μ, μ' es una función parcial de **direcciones** a **valores**. Notación

- **Reescribir**: $\mu[l \mapsto V]$ es el store resultante de **pisar** $\mu(l)$ con V .
- **Extender**: $\mu \oplus (l \mapsto V)$ es el **store extendido** resultante de ampliar μ con una nueva asociación $l \mapsto V$ (asumiendo $l \notin \text{Dom}(\mu)$).

Y las reducciones ahora toman la forma

$$M \mid \mu \rightarrow M' \mid \mu'.$$

En un paso posiblemente hay un nuevo store, porque puede haber habido una operación de asignación.

La intuición de la semántica es

$$\frac{l \notin \text{Dom}(\mu)}{\text{ref } V \mid \mu \rightarrow l \mid \mu \oplus (l \mapsto V)} (\text{E-RefV})$$

el efecto de hacer un $\text{ref } V$ es alocar una posición de memoria (meterlo en el store), creando una nueva dirección y asignándole el valor V .

Los valores posibles ahora incluyen las **direcciones**,

$$V ::= \dots \mid \text{unit} \mid \lambda x : \sigma. M \mid l$$

Dado que son un subconjunto de los términos, debemos ampliar los términos con direcciones. Estas son producto de la formalización y **no** se pretende que sean usadas por los programadores.

$$\begin{aligned}
 M ::= & x \\
 & | \lambda x : \sigma. M \\
 & | M \ N \\
 & | unit \\
 & | \text{ref } M \\
 & | !M \\
 & | M := N \\
 & | l
 \end{aligned}$$

Estaría bueno juntar esto para que quede todos los términos juntos y redactado de una forma que la tire de una en vez de ir explorando.

Juicios de tipado

Los valores que tienen las *locations* dependen de los valores que se almacenan en la dirección, una situación parecida a las variables libres. Entonces necesitamos un contexto de tipado para las direcciones. Σ va a ser una función parcial de direcciones en tipos.

$$\Gamma | \Sigma \triangleright M : \sigma$$

Y las reglas de tipado,

$$\frac{\Gamma | \Sigma \triangleright M_1 : \sigma}{\Gamma | \Sigma \triangleright \text{ref } M_1 : \text{Ref } \sigma} (\text{T-Ref})$$

$$\frac{\Gamma | \Sigma \triangleright M_1 : \text{Ref } \sigma}{\Gamma | \Sigma \triangleright !M_1 : \sigma} (\text{T-DeRef})$$

$$\frac{\Gamma | \Sigma \triangleright M_1 : \text{Ref } \sigma \quad \Gamma | \Sigma \triangleright M_2 : \sigma}{\Gamma | \Sigma \triangleright M_1 := M_2 : Unit} (\text{T-Assign})$$

$$\frac{\Sigma(l) = \sigma}{\Gamma | \Sigma \triangleright l : \text{Ref } \sigma} (\text{T-Loc})$$

Semántica operacional retomada

$$V ::= true \mid false \mid 0 \mid \underline{n} \mid unit \mid \lambda x : \sigma. M \mid l$$

Juicios de evaluación en un paso. Ahora van a tener la pinta

$$M \mid \mu \rightarrow M' \mid \mu'$$

$$\begin{array}{c}
\frac{M_1 \mid \mu \rightarrow M'_1 \mid \mu'}{!M_1 \mid \mu \rightarrow !M'_1 \mid \mu'} \text{(E-Deref)} \quad \frac{\mu(l) = V}{!l \mid \mu \rightarrow V \mid \mu} \text{(E-DerefLoc)} \\
\text{antes de hacer una desreferencia, necesito que el término llegue a un valor} \\
\frac{M_1 \mid \mu \rightarrow M'_1 \mid \mu'}{M_1 := M_2 \mid \mu \rightarrow M'_1 := M_2 \mid \mu'} \text{(E-Assign1)} \quad \frac{M_2 \mid \mu \rightarrow M'_2 \mid \mu'}{V := M_2 \mid \mu \rightarrow V := M'_2 \mid \mu'} \text{(E-Assign2)} \\
\frac{}{l := V \mid \mu \rightarrow \text{unit} \mid \mu[l \mapsto V]} \text{(E-Assign)} \\
\frac{M_1 \mid \mu \rightarrow M'_1 \mid \mu'}{\text{ref } M_1 \mid \mu \rightarrow \text{ref } M'_1 \mid \mu'} \text{(E-Ref)} \quad \frac{l \notin \text{Dom}(\mu)}{\text{ref } V \mid \mu \rightarrow l \mid \mu \oplus (l \mapsto V)} \text{(E-RefV)}
\end{array}$$

M puede ser complejo, así que hasta que llegue a un valor puede haber cambiado el store.

$$\begin{array}{c}
\frac{M_1 \mid \mu \rightarrow M'_1 \mid \mu'}{M_1 \ M_2 \mid \mu \rightarrow M'_1 \ M_2 \mid \mu'} \text{(E-App1)} \quad \frac{M_2 \mid \mu \rightarrow M'_2 \mid \mu'}{V_1 \ M_2 \mid \mu \rightarrow V_1 \ M'_2 \mid \mu'} \text{(E-App2)} \\
\frac{}{\lambda x : \sigma. M \ V \mid \mu \rightarrow M\{x \leftarrow V\} \mid \mu} \text{(E-AppAbs)}
\end{array}$$

El resto de las reglas son similares, pero no modifican el store.

El store se puede modificar en el medio, por ej si tengo

$$\begin{array}{c}
! \text{ref } V \mid \mu \rightarrow \text{(E-Deref, E-RefV)} \quad !l \mid \mu \oplus (l \mapsto V) \\
\rightarrow \text{(E-DerefLoc)} \quad V \mid \mu \oplus (l \mapsto V)
\end{array}$$

esto es fundamental ya que rompe la idea de funcional, en la que evolucionando un término no podía haber ningún *side effect*

Corrección

En la [section 1.2.3 Corrección](#) se habla de corrección en sistemas de tipos. Tenemos que reformularla en el marco de *referencias*.

Para la preservación nos gustaría definirlo como

$$\Gamma \mid \Sigma \triangleright M : \sigma \text{ y } M \mid \mu \rightarrow M' \mid \mu' \text{ entonces } \Gamma \mid \Sigma \triangleright M' : \sigma$$

pero esto tiene el problema de que nada fuerza la *coordinación* entre Σ y Γ . Por ejemplo, si tenemos

- $M = !l$
- $\Gamma = \emptyset$
- $\Sigma(l) = \text{Nat}$
- $\mu(l) = \text{true}$

entonces $\Gamma \mid \Sigma \triangleright M : \text{Nat}$ y $M \mid \mu \rightarrow \text{true} \mid \mu$ **pero** $\Gamma \mid \Sigma \triangleright \text{true} : \text{Nat}$ no vale.

Necesitamos un mecanismo para "**tipar**" los stores. Que haya una compatibilidad entre los stores y el contexto de tipado.

Def. 1.16 (Compatibilidad). Vamos a decir que un store es **compatible** con un juicio de tipado,

$$\Gamma|\Sigma \triangleright \mu \text{ sii}$$

1. $Dom(\Sigma) = Dom(\mu)$ y
2. $\Gamma|\Sigma \triangleright \mu(l) : \Sigma(l)$ para todo $l \in Dom(\mu)$

Reformulamos la preservación como

$$\Gamma|\Sigma \triangleright M : \sigma \text{ y } M \mid \mu \rightarrow N \mid \mu' \text{ y } \Gamma|\Sigma \triangleright \mu \text{ entonces } \Gamma|\Sigma \triangleright N : \sigma$$

esto es **casi** correcto, porque no contempla la posibilidad de que haya habido algún alloc en la reducción de M a N. Puede crecer en dominio

Def. 1.17 (Preservación para $\lambda^{\dots let}$). Si

- $\Gamma|\Sigma \triangleright M : \sigma$
- $M \mid \mu \rightarrow N \mid \mu'$ y
- $\Gamma|\Sigma \triangleright \mu$

implica que existe $\Sigma' \supseteq \Sigma$ tal que

- $\Gamma|\Sigma' \triangleright N : \Sigma$ y
- $\Gamma|\Sigma' \triangleright \mu'$

Def. 1.18 (Progreso para $\lambda^{\dots let}$). Si M es cerrado y bien tipado ($\emptyset|\Sigma \triangleright M : \sigma$ para algún Σ, σ) entonces:

1. M es un valor
2. o bien para cualquier store μ tal que $\emptyset|\Sigma \triangleright \mu$, existe M' y μ' tal que $M \mid \mu \rightarrow M' \mid \mu'$

Ejemplos

Falta copiar ejemplos:

Succ y refs

Let in

Ejemplo de que no todo termina

Importante: No todo término cerrado y bien tipado termina en λ^{bnr} (booleanos, naturales y registros).

1.3.5 Recursión

En programación funcional es muy común tener una función definida *recursivamente*,

$$f = f \dots f \dots f \dots$$

Términos y tipado

$$M ::= \dots \mid \text{fix } M$$

fix viene de la idea de **punto fijo**, aplicar la f varias veces.

No hacen falta nuevos tipos, pero si una regla de tipado

$$\frac{\Gamma \triangleright M : \sigma_1 \rightarrow \sigma_1}{\Gamma \triangleright \text{fix } M : \sigma_1} (\text{T-Fix})$$

M no puede ser cualquier cosa, tiene que ser una función que vaya del mismo dominio a la misma imagen.

Semántica operacional small-step

No hay valores nuevos, pero si reglas de eval en un paso nuevas.

$$\frac{M_1 \rightarrow M'_1}{\text{fix } M_1 \rightarrow \text{fix } M'_1} (\text{E-Fix})$$

$$\frac{}{\text{fix } (\lambda f : \sigma.M) \rightarrow M\{f \leftarrow \text{fix } (\lambda f : \sigma.M)\}} (\text{E-FixBeta})$$

La aplicación normal era

$$(\lambda x : \sigma.M) M_2 \rightarrow M\{x \leftarrow M_2\}$$

pero acá el x lo reemplazamos por el mismo fix.

Ejemplo: Factorial

Sea el término M

$$\begin{aligned} M &= \lambda f : \text{Nat} \rightarrow \text{Nat}. \\ &\quad \lambda x : \text{Nat}. \\ &\quad \text{if } \text{iszero}(x) \text{ then } \underline{1} \text{ else } x * f(\text{pred}(x)) \end{aligned}$$

M tiene tipo $(\text{Nat} \rightarrow \text{Nat}) \rightarrow (\text{Nat} \rightarrow \text{Nat})$. En

$$\text{let } fact : \text{Nat} \rightarrow \text{Nat} = \text{fix } M \text{ in } fact \underline{2}$$

$$\begin{aligned} \text{fix } M &= \text{fix } \lambda f. \lambda x. \text{if } \text{iszero}(x) \text{ then } \underline{1} \text{ else } x * (f \text{ pred}(x)) \\ &\rightarrow \lambda x. \text{if } \text{iszero}(x) \text{ then } 1 \\ &\text{else } x * ([\text{fix } \lambda f. \lambda x. \text{if } \text{iszero}(x) \text{ then } \underline{1} \text{ else } x * (f \text{ pred}(x))] \text{ pred}(x)) \\ &\rightarrow \lambda x. \text{if } \text{iszero}(x) \text{ then } 1 \\ &\text{else } x * \lambda x. \text{if } \text{iszero}(x) \text{ then } 1 \text{ else } x * ([\text{fix } \lambda f. \lambda x. \text{if } \text{iszero}(x) \text{ then } \underline{1} \text{ else } x * (f \text{ pred}(x))] \text{ pred}(x)) \\ &\rightarrow \dots \end{aligned}$$

Falta seguir con la reducción

Ejemplo: Suma

Sea el término M

$$\begin{aligned} M &= \lambda s : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}. \\ &\quad \lambda x : \text{Nat}. \\ &\quad \lambda y : \text{Nat}. \\ &\quad \text{if } \text{iszero}(x) \text{ then } y \text{ else } \text{succ}(s \text{ pred}(x) y) \end{aligned}$$

En

$$\text{let } suma = \text{fix } M \text{ in } suma \underline{2} \underline{3}$$

Letrec

letrec es una sintaxis alternativa para definir funciones recursivas,

$$\text{letrec } f : \sigma \rightarrow \sigma = \lambda x : \sigma. M \text{ in } N$$

Por ejemplo

$$\begin{aligned} \text{letrec } fact : Nat \rightarrow Nat = \\ \lambda x : Nat. \text{ if } iszero(x) \text{ then } \underline{1} \text{ else } x * fact(pred(x)) \\ \text{in } fact \underline{3} \end{aligned}$$

no es más que *syntactic sugar*. Se puede reescribir a partir de fix:

$$\text{let } f = \text{fix } (\lambda f : \sigma \rightarrow \sigma. \lambda x : \sigma. M) \text{ in } N$$

1.4 Inferencia de tipos

Consiste en transformar términos **sin** información de tipos o con información **parcial** en términos terminales **tipables**, con toda la información de tipos.

Para eso tenemos que *inferir* la parte de los tipos que nos faltan.

Es un beneficio, porque escribir tipos puede ser error prone y tedioso. El mecanismo que vamos a ver se realiza en tiempo de compilación y no en runtime.

Para comenzar con la inferencia vamos a denotar el conjunto de términos sin anotaciones de tipos por Λ (a diferencia de Λ_τ que sí), que son todos los mismos pero reemplazando $\lambda x : \sigma. M$ por $\lambda x. M$.

No van a funcionar las reglas de evaluación con Λ . Es importante inferir los tipos antes de hacer cualquier tipo de semántica, porque sino no tendríamos garantía de no estar evaluando cosas como *true false* que no tienen sentido.

1.4.1 Definición formal de inferencia

Para obtener estos tipos Λ , vamos a usar la función Erase.

Def. 1.19. (Erase) Llamamos $\text{Erase}(\cdot)$ a la función que dado un término de LC *elimina* las anotaciones de tipos de las abstracciones.

$\text{Erase}(\cdot) : \Lambda_\tau \rightarrow \Lambda$ se define de la manera esperada.

Ejemplo. $\text{Erase}(\lambda x : Nat. \lambda f : Nat \rightarrow Nat. f x) = \lambda x. \lambda f. f x$

Y así podemos definir formalmente el problema de la inferencia: Dado un término U **sin** anotaciones de tipo, hallar un término estándar (**con** anotaciones de tipo) M tal que

1. $\Gamma \triangleright M : \sigma$ para algún Γ, σ , y
2. $\text{Erase}(M) = U$

Ejemplos:

- Para $U = \lambda x.succ(x)$ tomamos $M = \lambda x : Nat.succ(x)$ (no hay otra posibilidad)
- Para $U = \lambda x.\lambda f.f\ x$ obtenemos infinitos M_s , $M_{\sigma,\tau} = \lambda x : \sigma.\lambda f : \sigma \rightarrow \tau.f\ x$

Podemos encontrar muchos M_s que nos sirvan.

- Para $U = x\ x$ no existe ningún término M que lo cumpla.

chequeo de tipos \neq inferencia de tipos.

El chequeo de tipos consistía en dado un término estándar M determinar si existían Γ y σ tales que $\Gamma \triangleright M : \sigma$ es tipable. El chequeo de tipos es guiado por sintaxis, la inferencia es más compleja.

por ejemplo en matemática, no es lo mismo demostrar un teorema que tener una demostración y verificar que sea correcta.

1.4.2 Variables de tipos

Motivación

Para formalizar la noción de que diferentes tipos pueden ser válidos para una inferencia, por ejemplo dado

$$\lambda x.\lambda f.f\ (f\ x)$$

para cada σ , M_σ es una solución posible

$$M_\sigma = \lambda x : \sigma.\lambda f : \sigma \rightarrow \sigma.f\ (f\ x)$$

Pero σ representa un tipo concreto. Para escribir una única expresión que englobe a todas, usamos **variables de tipo**: s es una variable de tipo que representa una expresión de tipos arbitraria ($Nat, Bool, Nat \rightarrow Bool$, etc.).

$$\lambda x : s.\lambda f : s \rightarrow s.f\ (f\ x)$$

la expresión no es una solución en si misma, pero la **sustitución** de s por cualquier expresión de tipos nos da una solución válida.

Def. 1.20 (Variables de tipo). Extendemos las **expresiones de tipo** del LC con **variables de tipo** s, t, u, \dots

$$\sigma ::= s \mid Nat \mid Bool \mid \sigma \rightarrow \tau$$

Denotamos con \mathcal{V} al conjunto de variables de tipo y \mathcal{T} al de tipos así definidos.

Ejemplo. Por ejemplo,

- $s \rightarrow t$
- $Nat \rightarrow Nat \rightarrow t$
- $Bool \rightarrow t$.

1.4.3 Sustitución

Una **sustitución** es una función que mapea variables de tipo en expresiones de tipo. Usamos S, T , para sustituciones. Formalmente, $S : \mathcal{V} \rightarrow \mathcal{T}$. La idea de la sustitución es instanciar las variables de tipos en tipos concretos, por ejemplo $Ss = Int$.

Solo nos interesan las S tales que $\{t \in \mathcal{V} \mid St \neq t\}$ (el soporte) es finito (este es un detalle técnico para que no sea infinita la sustitución)

Se pueden aplicar a,

1. Una expresión de tipos σ (escribimos $S\sigma$)
2. Un término M (escribimos SM)
3. Un contexto de tipado $\Gamma = \{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$

$$S\Gamma \stackrel{\text{def}}{=} \{x_1 : S\sigma_1, \dots, x_n : S\sigma_n\}$$

Notaciones

- El conjunto $\{t \mid St \neq t\}$ se llama **soporte** de S , representa las variables afectadas por S .
- Notamos $\{\sigma_1/t_1, \dots, \sigma_n/t_n\}$ para la sustitución con soporte $\{t_1, \dots, t_n\}$
- Id es la **sustitución identidad**, con soporte \emptyset (no afecta ninguna)

1.4.4 Instancia de un juicio de tipado

Def. 1.21 (Instancia de un juicio de tipado). Un juicio de tipado $\Gamma' \triangleright M' : \sigma'$ es una **instancia** de $\Gamma \triangleright M : \sigma$ si existe una sustitución de tipos S tal que

$$\Gamma' \supseteq S\Gamma, M' = SM \text{ y } \sigma' = S\sigma$$

Es agarrar y aplicar la sust S a todas las partes. Γ' puede ser un poco más grande, tiene que incluir.

Prop. 1.5. Si $\Gamma \triangleright M : \sigma$ es derivable, entonces cualquier instancia del mismo también lo es.

1.4.5 Función de inferencia $\mathbb{W}(\cdot)$

Vamos a definir una función $\mathbb{W}(\cdot)$ que dado un término U sin anotaciones de tipo, verifica

- **Corrección:** $\mathbb{W}(U) = \Gamma \triangleright M : \sigma$ implica
 - $\text{Erase}(M) = U$ y
 - $\Gamma \triangleright M : \sigma$ es derivable

Dado el U nos devuelve algo que chequea tipos y al borrarle los tipos nos devuelve U . I.e es correcto

- **Compleitud:** Si $\Gamma \triangleright M : \sigma$ es derivable y $\text{Erase}(M) = U$ entonces
 - $\mathbb{W}(U)$ tiene éxito, y
 - produce un juicio de tipado $\Gamma' \triangleright M' : \sigma'$ tal que $\Gamma \triangleright M : \sigma$ es instancia de él. *Se dice que $\mathbb{W}(\cdot)$ computa un **tipo principal**, el más general. Tal vez no instancia algunas cosas.*

Si existe ese M , queremos que el algoritmo de $\mathbb{W}(\cdot)$ ande. No necesariamente lo encuentra exactamente, pero al menos produce una instancia.

Tal vez sabemos que existe un juicio que tipa a un término como $\text{Nat} \rightarrow \text{Nat}$, pero en realidad se puede tipar como algo más general, como $s \rightarrow s$. El algoritmo debería devolver el más general y no la instancia.

1.4.6 Unificación

Motivación

Queremos definir $\mathbb{W}(\text{succ}(U)) \stackrel{\text{def}}{=} \text{succ}(U)$. Suponemos que $\mathbb{W}(U) = \Gamma \triangleright M : \tau$, y queremos saber si τ puede ser un *Nat*, si son *compatibles*.

Idea: yo tengo dos expr de tipos, y quiero ver si son compatibles, las puedo ver como iguales. Para eso tengo que sustituir adecuadamente. Si encuentro una sustitución que me da que son iguales, entonces son compatibles.

El proceso de determinar si existe una sustitución S tal que dos expresiones de tipos σ, τ son unificables (i.e $S\sigma = S\tau$) se llama **unificación**. Por ejemplo,

- El tipo $s \rightarrow t$ es **compatible** o **unificable** con $\text{Nat} \rightarrow u$? Si, porque podemos tomar la sustitución $S \stackrel{\text{def}}{=} \{\text{Nat}/s, u/t\}$.

Observar que $S(s \rightarrow t) = \text{Nat} \rightarrow u = S(\text{Nat} \rightarrow u)$ ya que S no afecta *Nat* ni u .

- El tipo s es **unificable** con *Nat*? Si, la sust S anterior es tal que $Ss = \text{Nat}$.

Composición de sustituciones

Def. 1.22 (Composicion de sustituciones). La **composición** de S y T , denotada $S \circ T$, es la sustitución que comporta

$$(S \circ T)(\sigma) = ST\sigma$$

Ejemplo 1.1. Sea $S = \{u \rightarrow \text{Bool}/t, \text{Nat}/s\}$ y $T = \{v \times \text{Nat}/u, \text{Nat}/s\}$ entonces $T \circ S = \{(v \times \text{Nat}) \rightarrow \text{Bool}/t, v \times \text{Nat}/u, \text{Nat}/s\}$

Decimos que $S = T$ si tienen el mismo soporte (variables afectadas) y $St = Tt$ para todo t en su soporte.

Prop. 1.6. Algunas props,

- $S \circ \text{Id} = \text{Id} \circ S = S$
- $S \circ (T \circ U) = (S \circ T) \circ U$ (asociatividad)
- La conmutatividad no vale.

Def. 1.23. Una sustitución S es **más general** que T si existe U tal que $T = U \circ S$.

La idea es que S es más general que T porque T se obtiene instanciando S .

Ejemplo 1.2. $S = \{t \rightarrow t/u\}$ es más general que $T = \{\text{Nat} \rightarrow \text{Nat}/u\}$, porque tomando $U = \{\text{Nat}/t\}$, $T = U \circ S$.

Lo que nos va a interesar en el algoritmo es obtener el tipo o sustitución más general. El que *instancie lo menos posible*.

MGU

Def. 1.24 (Ecuación de unificación). Una **ecuación de unificación** es una expresión de la forma $\sigma_1 \doteq \sigma_2$. Una sustitución S es la **solución** de conjunto de ecuaciones de unificación $\{\sigma_1 \doteq \sigma'_1, \dots, \sigma_n \doteq \sigma'_n\}$ si $S\sigma_1 = S\sigma'_1, \dots, S\sigma_n = S\sigma'_n$

Vamos a escribir ecuaciones con restricciones que quiero que se cumplan.

Ejemplo 1.3. Ejemplos,

- La sust $\{\text{Bool}/v, \text{Bool} \times \text{Nat}/u\}$ es solución de $\{v \times \text{Nat} \rightarrow \text{Nat} \doteq u \rightarrow \text{Nat}\}$ (reemplaza de ambos lados)

- $\{Bool \times Bool/v, (Bool \times Bool) \times Nat/u\}$ también
- $\{v \times Nat/u\}$ también, y es más general.
- $\{Nat \rightarrow s \doteq t \times u\}$ **no** tiene solución.
- $\{u \rightarrow Nat \doteq u\}$ no tiene solución. (Occurs check)

Def. 1.25 (MGU). Una sustitución S es un **MGU** (Unificador más general, most general unifier) de $\{\sigma_1 \doteq \sigma'_1, \dots, \sigma_n \doteq \sigma'_n\}$ si

1. Es solución de $\{\sigma_1 \doteq \sigma'_1, \dots, \sigma_n \doteq \sigma'_n\}$,
2. Es más general que cualquier otra solución de $\{\sigma_1 \doteq \sigma'_1, \dots, \sigma_n \doteq \sigma'_n\}$.

Ejemplo 1.4. La sustitución $\{Bool/v, Bool \times Nat/u\}$ solución de $\{v \times Nat \rightarrow Nat \doteq u \rightarrow Nat\}$ pero no es un MGU porque es instancia de $\{v \times Nat/u\}$, que si es MGU.

Algoritmo de unificación

Teorema 1.3. Si $\{\sigma_1 \doteq \sigma'_1, \dots, \sigma_n \doteq \sigma'_n\}$ tiene sol, existe un MGU y es único salvo renombre de variables.

El algoritmo va a tener,

- **Entrada:** Conjunto de ecuaciones de unificación $\{\sigma_1 \doteq \sigma'_1, \dots, \sigma_n \doteq \sigma'_n\}$
- **Salida:**
 - **MGU** S de $\{\sigma_1 \doteq \sigma'_1, \dots, \sigma_n \doteq \sigma'_n\}$, si tiene solución
 - **falla** en caso contrario.

Algoritmo de Martelli-Montanari

Es un algoritmo no determinístico, porque tiene un montón de **reglas de simplificación** que podemos aplicar en cualquier orden. Estas simplifican conjuntos de pares de tipos a unificar (los goals)

$$G_0 \mapsto G_1 \mapsto \dots \mapsto G_n$$

termina cuando llegamos al goal vacío. Algunos pasos de simplificación usan una sustitución que representa una *solución parcial* al problema,

$$G_0 \mapsto G_1 \mapsto_{S_1} G_2 \mapsto \dots \mapsto_{S_k} G_n$$

Si la secuencia es exitosa, el MGU es $S_k \circ \dots \circ S_1$

Reglas:

σ, τ tipos concretos y s var de tipo. C un constructor de tipos de cualquier aridad.

1. Descomposición

$$\{\sigma_1 \rightarrow \sigma_2 \doteq \tau_1 \rightarrow \tau_2\} \cup G \mapsto \{\sigma_1 \doteq \tau_1, \sigma_2 \doteq \tau_2\} \cup G$$

$$\{Nat \doteq Nat\} \cup G \mapsto G$$

$$\{Bool \doteq Bool\} \cup G \mapsto G$$

Caso general

$$\{C(\sigma_1, \dots, \sigma_n) \doteq C(\tau_1, \dots, \tau_n)\} \cup G \mapsto \{\sigma_1 \doteq \tau_1, \dots, \sigma_n \doteq \tau_n\} \cup G$$

2. **Eliminación de par trivial**

$$\{s \doteq s\} \cup G \mapsto G$$

3. **Swap**: si σ no es una variable,

$$\{\sigma \doteq s\} \cup G \mapsto \{s \doteq \sigma\} \cup G$$

4. **Eliminación de variable**: si $s \notin FV(\sigma)$,

$$\{s \doteq \sigma\} \cup G \mapsto_{\{\sigma/s\}} \{\sigma/s\}G$$

5. **Colisión**

$$\{\sigma \doteq \tau\} \cup G \mapsto \text{falla}$$

con $(\sigma, \tau) \in T \cup T^{-1}$ y $T = \{(Bool, Nat), (Nat, \sigma_1 \rightarrow \sigma_2), (Bool, \sigma_1 \rightarrow \sigma_2)\}$

(T^{-1} es una forma sucinta de invertir los pares)

Caso general: si $C \neq C'$ son constructores de tipo diferentes

$$\{C(\dots) \doteq C'(\dots)\} \cup G \mapsto \text{falla}$$

6. **Occur check**: si $s \neq \sigma$ y $s \in FV(\sigma)$

$$\{s \doteq \sigma\} \cup G \mapsto \text{falla}$$

Para la regla de descomposición de Nat y Nat, Bool y Bool podrían surgir por ej. de una descomposición de una func.

$$\{Nat \rightarrow t \doteq Nat \rightarrow s\} \mapsto \{Nat \doteq Nat, s \doteq t\} \mapsto \{s \doteq t\}$$

Ejemplo de secuencia exitosa

$$\begin{aligned} & \{(Nat \rightarrow r) \rightarrow (r \rightarrow u) \doteq t \rightarrow (s \rightarrow s) \rightarrow t\} \\ \mapsto^1 & \{Nat \rightarrow r \doteq t, r \rightarrow u \doteq (s \rightarrow s) \rightarrow t\} \\ \mapsto^3 & \{t \doteq Nat \rightarrow r, r \rightarrow u \doteq (s \rightarrow s) \rightarrow t\} \\ \mapsto^4_{Nat \rightarrow r/t} & \{r \rightarrow u \doteq (s \rightarrow s) \rightarrow (Nat \rightarrow r)\} \\ \mapsto^1 & \{r \doteq s \rightarrow s, u \doteq Nat \rightarrow r\} \\ \mapsto^4_{s \rightarrow s/r} & \{u \doteq Nat \rightarrow (s \rightarrow s)\} \\ \mapsto^4_{Nat \rightarrow (s \rightarrow s)/u} & \emptyset \end{aligned}$$

El MGU es

$$\begin{aligned} & \{Nat \rightarrow (s \rightarrow s)/u\} \circ \{s \rightarrow s/r\} \circ \{Nat \rightarrow r/t\} \\ = & \{Nat \rightarrow (s \rightarrow s)/t, s \rightarrow s/r, Nat \rightarrow (s \rightarrow s)/u\} \end{aligned}$$

Ejemplo de secuencia fallida

$$\begin{aligned} & \{r \rightarrow (s \rightarrow r) \doteq s \rightarrow ((r \rightarrow Nat) \rightarrow r)\} \\ \mapsto^1 & \{r \doteq s, s \rightarrow r \doteq (r \rightarrow Nat) \rightarrow r\} \\ \mapsto^4_{s/r} & \{s \rightarrow s \doteq (s \rightarrow Nat) \rightarrow s\} \\ \mapsto^1 & \{s \doteq s \rightarrow Nat, s \doteq s\} \\ \mapsto^6 & \text{falla} \end{aligned}$$

Teorema 1.4. El algoritmo de Martelli-Montanari siempre termina. Y sea G un conjunto de pares,

- si G tiene un unificador, el algoritmo termina exitosamente y retorna un MGU
- Si no tiene unificador, el algoritmo termina con **falla**

1.4.7 Algoritmo de inferencia

Importante: Tiene que implementar las reglas de tipado que *ya existen*. No tiene que hacer nuevas.

Términos simples

$$\begin{aligned}\mathbb{W}(0) &\stackrel{\text{def}}{=} \emptyset \triangleright 0 : Nat \\ \mathbb{W}(true) &\stackrel{\text{def}}{=} \emptyset \triangleright true : Bool \\ \mathbb{W}(false) &\stackrel{\text{def}}{=} \emptyset \triangleright false : Bool \\ \mathbb{W}(x) &\stackrel{\text{def}}{=} \{x : s\} \triangleright x : s, \text{ con } s \text{ variable fresca}\end{aligned}$$

Inferencia de succ

Sea $\mathbb{W}(U) = \Gamma \triangleright M : \tau, S = \text{MGU}(\{\tau \doteq Nat\})$,

$$\mathbb{W}(\text{succ}(U)) \stackrel{\text{def}}{=} S\Gamma \triangleright S \text{ succ}(M) : Nat$$

La idea para términos complejos es aplicar recursivamente el algoritmo para los sub-términos, y luego unificar los resultados.

similar para pred

Inferencia de iszero

Sea $\mathbb{W}(U) = \Gamma \triangleright M : \tau, S = \text{MGU}(\{\tau \doteq Nat\})$

$$\mathbb{W}(\text{iszero}(U)) \stackrel{\text{def}}{=} S\Gamma \triangleright S \text{ iszero}(M) : Bool$$

Inferencia de IfThenElse

Sean

- $\mathbb{W}(U) = \Gamma_1 \triangleright M : \rho$
- $\mathbb{W}(V) = \Gamma_2 \triangleright P : \sigma$
- $\mathbb{W}(W) = \Gamma_3 \triangleright Q : \tau$
- $S = \text{MGU}(\{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_i \text{ y } x : \sigma_2 \in \Gamma_j, i \neq j\} \cup \{\sigma \doteq \tau, \rho \doteq Bool\})$

Entonces

$$\begin{aligned}\mathbb{W}(\text{if } U \text{ then } V \text{ else } W) &\stackrel{\text{def}}{=} \\ S\Gamma_1 \cup S\Gamma_2 \cup S\Gamma_3 &\triangleright S(\text{if } M \text{ then } P \text{ else } Q) : S\sigma\end{aligned}$$

En un if, V y W tenían que tener el mismo tipo ($\sigma \doteq \tau$) y U tiene que ser algo compatible con bool ($\rho \doteq Bool$).

Además, Tenemos que pedir que los contextos de unificación sean compatibles. Por ej. si tenemos $x : Bool \in \Gamma_1$ y $x : s \in \Gamma_2$, podemos hacer $Bool/s$. Pero si tenemos cosas no compatibles como $x : Bool \in \Gamma_1$ y $x : Nat \in \Gamma_2$, no hay nada que se pueda hacer.

Para el tipo del if en el juicio, podría haber elegido tanto $S\sigma$ como $S\tau$ porque son iguales. Elegimos σ forma arbitraria.

Por qué unión de sustituciones y no sustitución de unión?

Cuál es la diferencia entre $S\Gamma_1 \cup S\Gamma_2$ y $S(\Gamma_1 \cup \Gamma_2)$?

Si bien el resultado final es el mismo, la segunda está **mal**, porque puede dar como resultado temporalmente algo que no cumple con el invariante de un contexto de tipado.

Por ejemplo $\{x : t\} \cup \{x : Nat\} = \{x : t, x : Nat\}$ que tiene más de un tipo para la misma variable. Luego de aplicar una sustitución pasarían a ser lo mismo, por ej. Nat/t .

Inferencia de aplicación

Sean

- $\mathbb{W}(U) = \Gamma_1 \triangleright M : \tau$
- $\mathbb{W}(V) = \Gamma_2 \triangleright N : \rho$
- $S = \text{MGU}(\{\sigma_1 \doteq \sigma_2 \mid x : \sigma_1 \in \Gamma_1 \text{ y } x : \sigma_2 \in \Gamma_2\} \cup \{\tau \doteq \rho \rightarrow t\})$, con t una variable fresca

entonces

$$\mathbb{W}(U \ V) \stackrel{\text{def}}{=} S\Gamma_1 \cup S\Gamma_2 \triangleright S(M \ N) : St$$

Es necesario aplicar la sustitución a t en el tipo del término porque si bien es fresca, dependiendo de como se unifique $\tau \doteq \rho \rightarrow t$, t podría pasar a ser otro tipo, por ejemplo uno concreto.

Inferencia de abstracción

Sea $\mathbb{W}(U) = \Gamma \triangleright M : \rho$. Tenemos dos posibilidades,

- Si el contexto de tipado tiene información sobre x (i.e $x : \tau \in \Gamma$ para algún τ), entonces

$$\mathbb{W}(\lambda x.U) \stackrel{\text{def}}{=} \Gamma \setminus \{x : \tau\} \triangleright \lambda x : \tau. M : \tau \rightarrow \rho$$

lo sacamos de Γ porque los contextos de tipado son para variables libres, y a x la vamos a ligar.

- Si no tiene (i.e $x \notin \text{Dom}(\Gamma)$) elegimos una variable fresca s y

$$\mathbb{W}(\lambda x.U) \stackrel{\text{def}}{=} \Gamma \triangleright \lambda x : s. M : s \rightarrow \rho$$

Por lo general va a ser el primer caso, que Γ venga con algún tipo para x . Por ejemplo si tenemos

$$\lambda x. \text{iszero}(x)$$

aplicar inferencia sobre $\text{iszero}(x)$ nos va a dar

$$\mathbb{W}(\text{iszero}(x)) = \{x : Nat\} \triangleright \text{iszero}(x) : Bool,$$

por lo que

$$\mathbb{W}(\lambda x. \text{iszero}(x)) = \emptyset \triangleright \lambda x : Nat. \text{iszero}(x) : Bool.$$

Se daría el segundo caso si en el cuerpo de la abstracción no se usa, por ejemplo para $\lambda x. \text{iszero}(0)$.

Otros ejemplos más sencillos para ver lo mismo son $\lambda x. x$ y $\lambda x. \text{true}$.

Definición alternativa de la práctica, mucho más sucinta y más fácil de entender:

$$\mathbb{W}(\lambda x.U) = \Gamma' \triangleright \lambda x : \tau_x. M : \tau_x \rightarrow \rho$$

- $\mathbb{W}(U) = \Gamma \triangleright M : \rho$
- $\Gamma' = \Gamma \ominus x$
- $\tau_x = \begin{cases} \alpha & \text{si } x : \alpha \in \Gamma \text{ (se usa } x \text{ en } U) \\ s \text{ fresca} & \text{si no} \end{cases}$

No se aplica ninguna sustitución porque no unificamos nada.

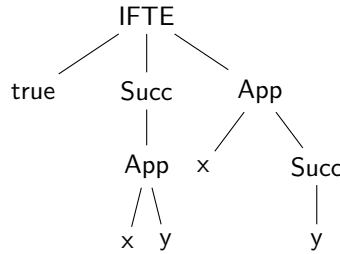
Inferencia de fix

Sean $\mathbb{W}(U) = \Gamma \triangleright M : \tau$, $S = \text{MGU}(\{\tau \doteq t \rightarrow t\})$ con t una variable fresca. Entonces,

$$\mathbb{W}(\text{fix } U) \stackrel{\text{def}}{=} S\Gamma \triangleright S\text{fix } (M) : St$$

Ejemplo de inferencia

Veamos la inferencia de `if true then succ(x y) else x (succ(y))`. Para ello es más fácil pensarlo sobre el AST, ya que el algoritmo es recursivo sobre la estructura sintáctica del término.



Para entender el algoritmo por primera vez está bueno hacerlo **top-down**. Pero es más difícil de seguir para ejecutarlo de forma práctica.

Para la práctica la recomendación es hacerlo **bottom-up**: numerar cada nodo del árbol sintáctico de las hojas a la raíz, y luego ejecutar en orden. En cada paso tenemos todos los resultados que necesitamos.

- $\mathbb{W}(\text{true}) = \emptyset \triangleright \text{true} : \text{Bool}$
- $\text{succ}(x \ y)$

$$\begin{aligned} \mathbb{W}(x) &= \{x : s\} \triangleright x : s \\ \mathbb{W}(y) &= \{y : t\} \triangleright y : t \\ \mathbb{W}(x \ y) &= \{x : t \rightarrow r, y : t\} \triangleright x \ y : r \\ &\quad \text{donde } S = \text{MGU}(\{s \doteq t \rightarrow r\}) = \{t \rightarrow r/s\} \\ \mathbb{W}(\text{succ}(x \ y)) &= \{x : t \rightarrow \text{Nat}, y : t\} \triangleright \text{succ}(x \ y) : \text{Nat} \\ &\quad \text{donde } S = \text{MGU}(\{r \doteq \text{Nat}\}) = \{\text{Nat}/r\} \end{aligned}$$

- $x \text{ (succ}(y))$

$$\begin{aligned}
\mathbb{W}(y) &= \{y : v\} \triangleright y : v \\
\mathbb{W}(\text{succ}(y)) &= \{y : \text{Nat}\} \triangleright \text{succ}(y) : \text{Nat} \\
&\text{donde } S = \text{MGU}(\{v \doteq \text{Nat}\}) = \{\text{Nat}/v\} \\
\mathbb{W}(x) &= \{x : u\} \triangleright x : u \\
\mathbb{W}(x \text{ succ}(y)) &= \{x : \text{Nat} \rightarrow w, y : \text{Nat}\} \triangleright x \text{ succ}(y) : w \\
&\text{donde } S = \text{MGU}(\{u \doteq \text{Nat} \rightarrow w\}) = \{\text{Nat} \rightarrow w/u\}
\end{aligned}$$

- $M = \text{if } \text{true} \text{ then } \text{succ}(x \ y) \text{ else } x \text{ (succ}(y))$

$$\begin{aligned}
\mathbb{W}(\text{true}) &= \emptyset \triangleright \text{true} : \text{Bool} \\
\mathbb{W}(\text{succ}(x \ y)) &= \{x : t \rightarrow \text{Nat}, y : t\} \triangleright \text{succ}(x \ y) : \text{Nat} \\
\mathbb{W}(x \ \text{succ}(y)) &= \{x : \text{Nat} \rightarrow w, y : \text{Nat}\} \triangleright x \ \text{succ}(y) : w \\
\mathbb{W}(M) &= \{x : \text{Nat} \rightarrow \text{Nat}, y : \text{Nat}\} \triangleright M : \text{Nat} \\
&\text{donde } S = \text{MGU}(\{t \rightarrow \text{Nat} \doteq \text{Nat} \rightarrow w, t \doteq \text{Nat}, \text{Nat} \doteq w\}) \\
&= \{\text{Nat}/t, \text{Nat}/w\}
\end{aligned}$$

$t \rightarrow \text{Nat} \doteq \text{Nat} \rightarrow w, t \doteq \text{Nat}$ salen de unificar contextos y $\text{Nat} \doteq w$ de que el tipo de ambas partes del if tiene que ser igual

Ejemplo de falla

$$M = \text{if } \text{true} \text{ then } x \ \underline{2} \text{ else } x \ \text{true}$$

$$\begin{aligned}
\mathbb{W}(x) &= \{x : s\} \triangleright x : s \\
\mathbb{W}(\underline{2}) &= \emptyset \triangleright \underline{2} : \text{Nat} \\
\mathbb{W}(x \ \underline{2}) &= \{x : \text{Nat} \rightarrow t\} \triangleright x \ \underline{2} : t
\end{aligned}$$

$$\begin{aligned}
\mathbb{W}(x) &= \{x : u\} \triangleright x : u \\
\mathbb{W}(\text{true}) &= \emptyset \triangleright \text{true} : \text{Bool} \\
\mathbb{W}(x \ \text{true}) &= \{x : \text{Bool} \rightarrow v\} \triangleright x \ \text{true} : v
\end{aligned}$$

$$\mathbb{W}(M) = \text{falla}$$

porque no existe el $\text{MGU}(\{\text{Nat} \rightarrow t \doteq \text{Bool} \rightarrow v\})$.

1.5 Subtipado

Diego: lo más relevante es el contexto, algunos detalles no son tan importantes.

Motivación

El sistema de tipos que estudiamos descarta programas incorrectos, pero también algunos "buenos"

$$(\lambda x : \{a : \text{Nat}\}.x.a) \{a = 1, b = \text{true}\}$$

este término no tipa porque tiene que ser exactamente $\{a : \text{Nat}\}$, pero no tendría nada de malo que tenga cosas demás.

Cuando uno tipa de alguna manera lo que quiere evitar es darle semántica a cosas que no tienen sentido, pero estamos rechazando cosas que en runtime andarían bien.

1.5.1 Principio de substitutividad

Es una relación de dos tipos,

$$\sigma <: \tau$$

Se lee como “En todo contexto donde se espera una expresión de tipo τ , puede usarse una de tipo σ en su lugar **sin** que ello genere un error”.

Otras formas de verlo:

- Podemos decir que σ es un **subtipo** de τ y τ **supertipo** de σ .
- Todo valor descrito por σ también es descrito por τ
- Los elementos de σ son un subconjunto de los elementos de τ .

Y se refleja con una nueva regla de tipado llamada **subsumption**

$$\frac{\Gamma \triangleright M : \sigma \quad \sigma <: \tau}{\Gamma \triangleright M : \tau} \text{(T-Subs)}$$

Ejemplos:

- $M : \text{Nat}$ y se que $\text{Nat} <: \text{Int}$ puedo ver a $M : \text{Int}$.
- $M : \text{Perro}$ y se que $\text{Perro} <: \text{Animal}$ (perro es un subtipo de animal), entonces puedo ver a $M : \text{Animal}$.

Antes nuestro sistema de tipos estaba guiado por sintaxis, pero ahora tenemos algo que no necesariamente viene de ahí.

1.5.2 Reglas de subtipado

Para los tipos base asumimos que nos informan de que manera están relacionados, son como reglas *axiomáticas*

$$\frac{}{\text{Nat} <: \text{Float}} \text{(S-NatFloat)} \quad \frac{}{\text{Int} <: \text{Float}} \text{(S-IntFloat)}$$

$$\frac{}{\text{Bool} <: \text{Nat}} \text{(S-BoolNat)}$$

porque *Bool* se puede tomar como 0 y 1

Además, podemos definir un pre orden a partir de la relación de subtipado

$$\frac{}{\sigma <: \sigma} \text{(S-Refl)} \quad \frac{\sigma <: \tau \quad \tau <: \rho}{\sigma <: \rho} \text{(S-Trans)}$$

por ej.

$$\frac{\frac{\checkmark}{\text{Nat} <: \text{Int}} \quad \frac{\checkmark}{\text{Int} <: \text{Float}}}{\text{Nat} <: \text{Float}} \text{(S-Trans)}$$

1.5.3 Subtipado en $\lambda^{\dots r}$

(ver [subsection 1.3.2 \$\lambda^{\dots r}\$ - Registros](#))

Subtipado a lo ancho

$$\frac{}{\{ l_i : \sigma_i \mid i \in 1..n+k \} <: \{ l_i : \sigma_i \mid i \in 1..n \}} \text{(S-RcdWidth)}$$

Ejemplo

$$\{ \text{nombre} : \text{String}, \text{edad} : \text{Int} \} <: \{ \text{nombre} : \text{String} \}$$

si tengo un registro y lo extiendo con más cosas lo tomo como subtipo

■ Rcd es por ReCoRD

Observaciones,

- $\sigma <: \{ \}$ para todo σ record (cualquiera es más grande).
- No hay tipo τ tal que $\tau <: \sigma$ para todo σ , siempre puedo tomar un σ que tenga algún elemento de un tipo diferente. En general va a haber alguno que sea más chico que todos, a menos que lo definamos a mano.

Subtipado a lo profundo

$$\frac{\sigma_i <: \tau_i \quad i \in I = \{1..n\}}{\{ l_i : \sigma_i \}_{i \in I} <: \{ l_i : \tau_i \}_{i \in I}} \text{(S-RcdDepth)}$$

Ejemplo

$$\{ a : \text{Nat}, b : \text{Int} \} <: \{ a : \text{Float}, b : \text{Int} \}$$

Ejemplo

$$\frac{\frac{}{\{ a : \text{Nat}, b : \text{Nat} \} <: \{ a : \text{Nat} \}} \text{(S-RcdWidth)} \quad \frac{}{\{ m : \text{Nat} \} <: \{ \}} \text{(S-RcdWidth)}}{\frac{}{\{ x : \{ a : \text{Nat}, b : \text{Nat} \}, y : \{ m : \text{Nat} \} \} <: \{ x : \{ a : \text{Nat} \}, y : \{ \} \}} \text{(S-RcdDepth)}}$$

Permutaciones de campos

Los registros son descripciones de campos, y no deberían depender del orden dado. Hasta ahora, $\{ a : \text{Int}, b : \text{String} \} \neq \{ b : \text{String}, a : \text{Int} \}$ pero querríamos que $\{ a : \text{Int}, b : \text{String} \} <: \{ b : \text{String}, a : \text{Int} \}$ (y también vale al revés)

$$\frac{\{ k_j : \sigma_j \mid j \in 1..n \} \text{ es permutación de } \{ l_i : \tau_i \mid i \in 1..n \}}{\{ k_j : \sigma_j \mid j \in 1..n \} <: \{ l_i : \tau_i \mid i \in 1..n \}} \text{(S-RcdPerm)}$$

S-RcdPerm combinado con S-RcdWidth y S-Trans puede eliminar campos en cualquier parte de un registro, generando un supertipo (reordenando para que queden últimas las que quiero eliminar)

Regla combinada

$$\frac{\{ l_i \mid i \in 1..n \} \subseteq \{ k_j \mid j \in 1..m \} \quad k_j = l_i \Rightarrow \sigma_j <: \tau_i}{\{ k_j : \sigma_j \mid j \in 1..n \} <: \{ l_i : \tau_i \mid i \in 1..m \}} \text{(S-Rcd)}$$

n y m permiten Width, que cuando coinciden sean subtipos Depth y \subseteq Perm (además hace implícito que $m \geq n$)

1.5.4 Subtipado de funciones

Si tengo una función que espera recibir números positivos, entonces le voy a estar pasando todos los positivos. Si me pasan una que además se banca más, entonces mejor, no la voy a romper. Pero si me pasan una que se banca menos que los positivos, la voy a romper con algunos argumentos.

Una buena forma de pensar intuitivamente la covarianza y contravarianza es hacer ejemplos con registros, que tienen la interpretación más intuitiva de subtipado.

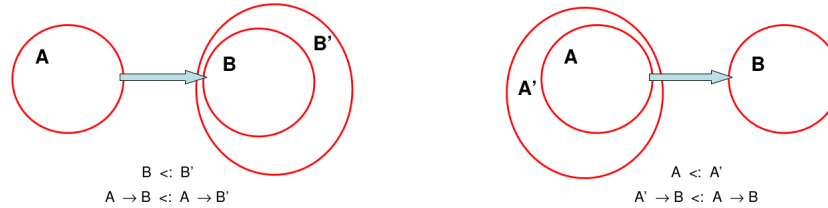


Figure 1.1: Esquema de Perla para entender el subtipado de funciones

$$\frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \rightarrow \tau <: \sigma' \rightarrow \tau'} (\text{S-Arrow} \mid \text{S-Func})$$

Para reemplazar una función por otra, tiene que

- Bancarse todos los argumentos, o más (contravarianza)
- El resultado tiene que ser reemplazable (covarianza)

Se dice que el constructor de tipos de función es **contravariante** en el primer argumento (dominio) y **covariante** en el segundo (imagen).

1.5.5 Subtipado de referencias

Ref es **invariante**, solo se comparan referencias de tipos equivalentes

$$\frac{\sigma <: \tau \quad \tau <: \sigma}{\text{Ref } \sigma <: \text{Ref } \tau} (\text{S-Ref})$$

no hace falta que sean iguales, como con los registros y las permutaciones.

Justificación

Es ref covariante?

$$\frac{\sigma <: \tau}{\text{Ref } \sigma <: \text{Ref } \tau} (\text{S-Ref})$$

Si tengo $r : \text{Ref Nat}$ y hago $!r$, que le puedo pasar? Y la escritura?

Si fuera covariante, uno esperaría que como $\text{Int} <: \text{Float}$, $\text{Ref Int} <: \text{Ref Float}$. Pero si tengo

```
let r = ref 3 // r : Ref Int
in
  r := 2.1; // Ref Int <: Ref Float, T-Sub, r: Ref Float
  !r
  // r: Int
  // Pero 2.1 no es int!
```

se rompe con la asignación.

(el 2 es un int que lo puedo ver como 2.0, float, pero el 2.1 es un float y no lo puedo ver como int)

Es ref contravariante?

$$\frac{\sigma <: \tau}{\text{Ref } \tau <: \text{Ref } \sigma} \text{(S-Ref)}$$

Como $\text{Int} <: \text{Float}$ y suponemos ref contravariante, $\text{Ref Float} <: \text{Ref Int}$

```
let r = ref 2.1 // r: Ref Float
in
  !r
  // Por Ref Float <: Ref Int y T-Sub, r: Ref Int
  // r: Int
  // Pero 2.1 no es Int!
```

Refinado de Ref

Para permitir algún tipo de subtipado, se agregan nuevas clases referencias de solo lectura y solo escritura. Source σ de lectura y Sink σ de escritura.

$$\frac{\Gamma|\Sigma \triangleright M : \text{Source } \sigma}{\Gamma|\Sigma \triangleright !M : \sigma} \quad \frac{\Gamma|\Sigma \triangleright M : \text{Sink } \sigma \quad \Gamma|\Sigma \triangleright N : \sigma}{\Gamma|\Sigma \triangleright M := N : \text{Unit}}$$

- Source (lectura) es covariante

$$\frac{\sigma <: \tau}{\text{Source } \sigma <: \text{Source } \tau} \text{(S-Source)} \quad \frac{\text{Int} <: \text{Float}}{\text{Source Int} <: \text{Source Float}}$$

$!r$ puede verse como Float aunque r sea de tipo Source Int .

Si tengo un ref 3 y lo desreferencio, puedo verlo como int o float 3.0

```
let r = ref 3
in
  !r // por Source Int <: Source Float
  :: Float
```

Si espero leer una ref a T, puedo esperar una ref a un tipo más bajo, más informativo.

- Sink (escritura) es contravariante.

$$\frac{\tau <: \sigma}{\text{Source } \sigma <: \text{Source } \tau} \text{(S-Sink)} \quad \frac{\text{Int} <: \text{Float}}{\text{Sink Float} <: \text{Sink Int}}$$

Por ejemplo,

```
let r = ref 2.1
in
  r := 3; // Usando Sink Float <: Sink Int
  !r
  :: Float
```

Puedo coercionar 3 a 3.0 y no se rompe nada.

Si espero escribir sobre una Ref a T, puedo esperar una Ref a un tipo más alto, menos informativo.

Se pueden relacionar con Ref,

$$\frac{}{\text{Ref } \tau <: \text{Source } \tau} (\text{S-RefSource}) \quad \frac{}{\text{Ref } \tau <: \text{Sink } \tau} (\text{S-RefSink})$$

1.5.6 Algoritmo

Muy lindo todo, pero como lo usamos para tipar? Hasta ahora nuestras reglas eran dirigidas por sintaxis, por lo que es inmediato implementar un algoritmo de chequeo de tipos a partir de ellas.

Pero con T-Subs, está guiada por la *oportunidad del tipo*, la podés aplicar cuando quieras como convenga. Esto hace que **no sea evidente como implementar un algoritmo** de chequeo de tipos a partir de las reglas, que no sea determinístico. Nos podemos sacar de encima el problema?

Propuesta N°1 de sistema: cambio en la regla de aplicación

La única regla en la que realmente hace falta subtipar es en la aplicación. Definimos una variante del sistema de tipado dirigida por sintaxis y la notamos con \mapsto .

$$\frac{\Gamma \mapsto M : \sigma \rightarrow \tau \quad \Gamma \mapsto N : \rho \quad \rho <: \sigma}{\Gamma \mapsto M N : \tau} (\text{T-App})$$

(y cambiando el resto del sistema para que use el tipado alternativo \mapsto)

Prop. 1.7. Se puede probar por inducción en las reglas de tipado que

1. $\Gamma \mapsto M : \sigma$ implica que $\Gamma \triangleright M : \sigma$
2. $\Gamma \triangleright M : \sigma$ implica que existe τ tal que $\Gamma \mapsto M : \tau$ con $\tau <: \sigma$.

Pero nos falta cubrir como implementar la relación $<:$. (recordar [subsection 1.5.2 Reglas de subtipado](#) y S-Rcd). Las reglas S-Refl y S-Trans no están guiadas por la sintaxis. Para sacarlas,

■ **Diego:** Lo que sigue ahora es algo anecdótico

- S-Refl está para resolver cosas de la pinta $\text{Nat} <: \text{Nat}$, entonces podemos remover la necesidad de la reflexividad agregando axiomas para cada tipo: $\text{Nat} <: \text{Nat}$, $\text{Float} <: \text{Float}$, $\text{Bool} <: \text{Bool}$.

Esto se puede probar en general pero no lo probamos

- Con un argumento similar, se puede demostrar que no es necesaria S-Trans agregando todas las combinaciones.

Sacando esas dos reglas, la parte de subtipado puede ser guiada por sintaxis? Si, el algoritmo es el siguiente

```
subtype(S, T) {
  // Si es un axioma, sabemos que si. Sino,
  // func
  if S == S1 -> S2 and T == T1 -> T2:
    return subtype(T1, S2) and subtype(S2, T2)

  // reg
  if S == {kj: Sj, j in 1..m} and T == {li: Ti, i in 1..n}:
    return {li, i in 1..n} subseteq {kj, j in 1..m} and
```

```
        forall i exists j kj = li and subtype(Sj, Ti)
    return false
}
```

Chapter 2

Paradigma de objetos

El modelo de cómputo que está detrás de POO, también puede caer diseño orientado a objetos pero es mucho más complejo. Acá vamos a modelar una parte chiquita.

2.1 Programación Orientada a Objetos

Conceptos y metáfora

- Todo programa es una simulación, y cada entidad del sistema siendo simulado se representa en el programa a través de un **objeto**
- Los objetos son la forma de abstraer un concepto físico o conceptual del mundo real
- El modelo de cómputo consiste en envío de mensajes: un sistema está formado por objetos que *colaboran* entre sí mediante mensajes.
- Los **mensajes** son solicitudes para que un objeto lleve a cabo operación. El **receptor** (el objeto que lo recibe) decide como llevarla a cabo, cuya implementación está descrita por un **método**.
- El conjunto de mensajes que responde un objeto se denomina **interfaz** o **protocolo**.
- Los objetos pueden tener **estado** interno que altere el comportamiento de los métodos. Se representa a través de un conjunto de **colaboradores internos** (también llamados **atributos** o **variables instancia**)

Ejemplo:

```
unRectangulo
  interfaz: area
  atributos: alto y ancho
  método: area = function() { return alto * ancho }
```

La única manera de interactuar con un objeto es a través de su protocolo. Su implementación no puede depender de detalles de implementación de otros objetos (principio heredado de TADs)

Def (Principio de ocultamiento de la información). El estado de un objeto es **privado** y solamente puede ser consultado o modificado por sus métodos. (*No todos los lenguajes imponen esta restricción*)

2.1.1 Method dispatch

Cómo hacemos por atrás para saber qué método de un objeto ejecutar cuando le llega un mensaje? El proceso que establece la asociación mensaje-método a ejecutar se llama **method dispatch**.

Si se hace en tiempo de *compilación* (se puede determinar a partir del código fuente) es **method dispatch estático**. En cambio, si se hace en runtime es **method dispatch dinámico**.

2.1.2 Corrientes

Quien es responsable de conocer los métodos de los objetos? Hay dos alternativas conocidas: **clasificación** y **prototipado**

2.1.3 Clasificación

Es la más mainstream. Las clases modelan **conceptos abstractos** del dominio de problema. Definen el comportamiento y la forma de un conjunto de objetos que instancian (sus **instancias**). Todo objeto es instancia de alguna clase. Son templates que tienen métodos, atributos y después se usan para instanciar objetos concretos.

Tienen

- Nombre
- Definición de variables de instancia
- Métodos de instancia. Por cada uno nombre, parámetros y cuerpo.

```
clase Point
Variables de instancia 'xCoord' e 'yCoord'
Métodos
x
  ^xCoord
y
  ^yCoord

dist: aPoint
  "Answer the distance between aPoint and the receiver."
  | dx dy |
  dx := aPoint x - xCoord.
  dy := aPoint y - yCoord.
  ^ (dx * dx + (dy * dy)) sqrt
```

Figure 2.1: Ejemplo de clase en sintaxis de Smalltalk

<pre> class INode Métodos de clase l: leftchild r:rightchild "Creates an interior node" ... Vars. de instancia 'left right' Métodos de instancia sum ^ left sum + right sum </pre>	<pre> class Leaf Métodos de clase new: anInteger "Creates a leaf" ... Vars. de instancia 'value' Métodos de instancia sum ^value </pre>
---	--

Ejemplos

- 1) Leaf new: 5
- 2) (INode l: (Leaf new: 3) r: (Leaf new: 4)) sum

Figure 2.2: Ejemplo de clase Node en sintaxis de Smalltalk

Self y super

self es una pseudovariable que durante la ejecución de un método referencia al receptor del mensaje. Se liga automáticamente y no puede ser asignada.

<pre> class INode Métodos de clase l: leftchild r:rightchild "Creates an interior node" ... Var. de instancia 'left right' Métodos de instancia l ^left r ^right sum ^ (self l) sum + (self r) sum </pre>
--

Figure 2.3: Ejemplo uso de self en Smalltalk

super es otra pseudovariable que referencia al objeto que recibe el mensaje. Cambia el proceso de activación al momento del envío de un mensaje.

Una expresión de la forma `super msg` que aparece en el cuerpo de un método `m` provoca que el **method lookup** se haga desde el padre de la clase anfitriona de `m`.

```

Object subclass: #C1
  Métodos de instancia
  m1
    ^self m2
  m2
    ^13

C1 subclass: #C2
  Métodos de instancia
  m1
    ^22
  m2
    ^23
  m3
    ^super m1

C2 subclass: #C3
  Métodos de instancia
  m1
    ^32
  m2
    ^33

(C2 new) m3. ----¿ Qué valor devuelve? 23
(C3 new) m3. ----¿ Qué valor devuelve? 33

```

Figure 2.4: Ejemplo super y self

Jerarquía de clases

Es común que nuevas clases se definan como *extensiones* de clases existentes, agregando o cambiando el comportamiento de algunos métodos y agregando nuevas variables de instancia o clase. Por eso, una clase puede **heredar de** o **extender** de una clase existente (llamada **superclase**). La transitividad de esta relación induce nociones de **ancestros** y **descendientes**.

Hay dos tipos

- **Simple:** una clase tiene un único padre (salvo la raíz). Esta es la que usan la mayoría de los lenguajes OO.
- **Múltiple:** una clase puede tener más de una clase padre.

Complica el proceso de method dispatch, ya que si tengo un método *m* definido en más de una superclase, ¿cual uso? Hay dos soluciones posibles:

- Establecer un *orden de búsqueda* sobre las superclases
- Pedir que se *redefinan* en la clase nueva todos los métodos que estén en más de una clase padre.

<pre>Object subclass: #Point instanceVarNames: 'xCoord yCoord' Métodos de clase x: p1 y: p2 ^self new setX: p1 setY: p2 Métodos de instancia x ^xCoord y ^yCoord setX: xValue setY: yValue xCoord := xValue. yCoord := yValue.</pre>	<pre>Point subclass: #ColorPoint instanceVarNames: 'color' Métodos de clase x: p1 y: p2 color: aColor instance instance := self x: p1 y: p2. instance color: aColor. ^instance Métodos de instancia color: aColor color := aColor color ^color</pre>
---	---

USO

```
ColorPoint x: 10 y: 20 color: red.
```

Figure 2.5: Ejemplo de herencia en Smalltalk

2.1.4 Prototipado

Construye instancias concretas que se interpretan como representantes canónicos de instancias (llamados prototipos). El resto de las instancias se generan clonando los prototipos (de forma shallow). Y los clones se pueden cambiar.

Ejemplo de creado de objeto en js

```
let celda = {
  contenido: 0,
  get: function() { return this.contenido; },
  set: function(n) { this.contenido = n; }
}

// Generar objetos
Celda = function() {
  this.contenido = 0;
  this.get = function() { return this.contenido; };
  this.set = function(n) { this.contenido = n; };
}

otracelda = new Celda();
```

2.2 Cálculo de objetos

Acá vamos a hacer semántica big step en vez de small step. De un gran paso llegás al valor. Vamos a ver un cálculo de objetos no tipado basado en Abadi y Cardelli, 98 que se llama ζ calculo (sigma pero una sigma cheta)

Ingredientes

- Los objetos son la única estructura computacional
- Los objetos son registros que tienen métodos como atributos (un campo normal va a ser un método que devuelve siempre lo mismo)

- Cada método tiene una única variable ligada que representa a self (o this) y un cuerpo que produce el resultado, que puede depender o no de self.
- Proveen dos operaciones: envío de mensaje (invocar un método) o redefinición de un método.

2.2.1 Sintaxis

$o, b ::= x$	<i>variable</i>
$ [l_i = \varsigma(x_i)b_i \text{ } i \in 1..n]$	<i>objeto</i>
$ o.l$	<i>selección / envío de mensaje</i>
$ o.l \Leftarrow \varsigma(x)b$	<i>redefinición de método</i>

Ejemplo.

$$o \stackrel{\text{def}}{=} [l_1 = \varsigma(x_1)[], l_2 = \varsigma(x_2)x_2.l_1]$$

- l_1 retorna el objeto vacío
- l_2 envía el mensaje l_1 a self (representado por el parámetro x_2)

2.2.2 Atributos vs métodos

El cálculo ς no incluye explícitamente atributos (campos), sino que se representan como métodos que no usan al parámetro self. De esta manera, el envío de un mensaje representa también a la selección de un atributo y la redefinición de un método representa también la asignación de un atributo

Como abusos de notación, vamos a

- Usar $[\dots, l = b, \dots]$ en vez de $l = \varsigma(x)b$ cuando no se usa x en b ,
- Usar $o.l := b$ en vez de $o.l \Leftarrow \varsigma(x)b$ cuando x no se usa en b .

2.2.3 Variables libres

ς es un ligador del parámetro self x_i en el cuerpo b_i de la expresión $\varsigma(x_i)b_i$.

■ Esto es similar a lc, probablemente en semántica vamos a tener que hacer una sustitución para ligarla.

Def. 2.1 (Variables libres).

$$\begin{aligned}
 \text{fv}(\varsigma(x)b) &= \text{fv}(b) \setminus \{x\} \\
 \text{fv}(x) &= \{x\} \\
 \text{fv}([l_i = \varsigma(x_i)b_i \text{ } i \in 1..n]) &= \bigcup_{i \in 1..n} \text{fv}(\varsigma(x_i)b_i) \\
 \text{fv}(o.l) &= \text{fv}(o) \\
 \text{fv}(o.l \Leftarrow \varsigma(x)b) &= \text{fv}(o.l) \cup \text{fv}(\varsigma(x)b)
 \end{aligned}$$

Un término o es **cerrado** si $\text{fv}(o) = \emptyset$.

2.2.4 Sustitución

$$\begin{aligned}
x\{c/x\} &= c \\
y\{c/x\} &= y \\
([l_i = \varsigma(x_i)b_i]^{i \in 1..n})\{c/x\} &= [l_i = (\varsigma(x_i)b_i)\{c/x\}]^{i \in 1..n} \\
(o.l)\{c/x\} &= o\{x/x\}.l \\
o.l \Leftarrow \varsigma(y)b\{c/x\} &= (o\{c/x\}).l \Leftarrow ((\varsigma(y)b)\{c/x\}) \\
(\varsigma(y)b)\{c/x\} &= \varsigma(y')b\{y'/y\}\{c/x\} \\
&\text{con } y' \notin \text{fv}(\varsigma(y)b) \cup \text{fv}(c) \cup \{x\}
\end{aligned}$$

Acomodamos las cosas para que no haya interferencias en la sustitución como clashing de nombres (primer reemplazo) y una vez que tengamos eso, aplicamos la sustitución que queremos.

2.2.5 Equivalencia de términos (\equiv)

Los términos $\varsigma(x)b$ y $\varsigma(y)(b\{y/x\})$ con $y \notin \text{fv}(b)$ se consideran equivalentes (α -conversión).

También, dos objetos que difieren en el orden de sus componentes son considerados equivalentes.

$$[l_1 = \varsigma(x_1)[\], l_2 = \varsigma(x_2)x_2.l_1] \equiv [l_2 = \varsigma(x_3)x_3.l_1, l_1 = \varsigma(x_1)[\]]$$

2.2.6 Semántica operacional

En lc, las reducciones llevaban de $M \rightarrow M'$ y eventualmente aplicando muchas reducciones llegábamos a un valor. Acá vamos a llegar en una sola.

Los valores van a ser objetos

$$v ::= [l_i = \varsigma(x_i)b_i]^{i \in 1..n}$$

Y vamos a aplicar reducciones **big-step** \longrightarrow , que en un paso de reducción pasan de una expresión a un valor.

$$\begin{aligned}
&\frac{}{v \longrightarrow v} [\text{Obj}] \\
&\frac{o \longrightarrow v' \quad v' \equiv [l_i = \varsigma(x_i)b_i]^{i \in 1..n} \quad b_j\{v'/x_j\} \longrightarrow v \quad j \in 1..n}{o.l_j \longrightarrow v} [\text{Sel}] \\
&\frac{o \longrightarrow [l_i = \varsigma(x_i)b_i]^{i \in 1..n} \quad j \in 1..n}{o.l_j \Leftarrow \varsigma(x)b \longrightarrow [l_j = \varsigma(x)b, l_i = \varsigma(x_i)b_i]^{i \in 1..n - \{j\}}} [\text{Upd}]
\end{aligned}$$

En Sel reduzco o hasta un valor v' para saber quien es self, ligo self en b_j que es el cuerpo del método correspondiente a l_j , y el resultado es reducir eso a un valor

Ejemplo. Reducción de $o = [a = [], l = \varsigma(x)x.a].l$

$$\begin{array}{c}
\frac{}{o \rightarrow o} [\text{Obj}] \quad \frac{}{= []} [\text{Obj}] \\
\frac{}{o \rightarrow o} [\text{Obj}] \quad \frac{\frac{}{= []} [\text{Obj}]}{[] \{o/x\} \rightarrow []} [\text{Sel}] \\
\frac{}{o \rightarrow o} [\text{Obj}] \quad \frac{\frac{}{= o.a}}{(x.a) \{o/x\} \rightarrow []} [\text{Sel}] \\
\frac{}{[a = [], l = \varsigma(x)x.a].l \rightarrow []} [\text{Sel}] \\
\hline
o
\end{array}$$

Ejemplo. Reducción de $([a = [], l = \varsigma(x)x.a].l \leftarrow \varsigma(y)[]) . l$

$$\begin{array}{c}
\frac{}{o \rightarrow o} [\text{Obj}] \quad \frac{}{= []} [\text{Obj}] \\
\frac{u \rightarrow [a = [], l = []]}{[a = [], l = \varsigma(x)x.a].l \leftarrow \varsigma(y)[] . l \rightarrow []} [\text{Upd}] \quad \frac{\frac{}{= []} [\text{Obj}]}{[] \{u/x\} \rightarrow []} [\text{Sel}] \\
\frac{}{([a = [], l = \varsigma(x)x.a].l \leftarrow \varsigma(y)[]) . l \rightarrow []} [\text{Sel}] \\
\hline
u
\end{array}$$

Ejemplo. Podemos tener problemas, por ej. el siguiente tiene una reducción infinita $\underbrace{[a = \varsigma(x)x.a].a}_o$

es como una llamada recursiva, pero sin caso base. La evaluación de esta expresión se indefinire. Es análogo a $\text{fix } \lambda x : \sigma.x$.

$$\begin{array}{c}
\frac{}{o \rightarrow o} [\text{Obj}] \quad \frac{\vdots}{= o.a} [\text{Sel}] \\
\frac{}{o \rightarrow o} [\text{Obj}] \quad \frac{\frac{}{= o.a}}{x.a \{o/x\} \rightarrow} [\text{Sel}] \\
\hline
o.a \rightarrow
\end{array}$$

2.2.7 Ejemplo: Naturales

Vamos a asumir que existen los objetos `true` y `false` que corresponden a booleanos.

Se podrían definir estilo Smalltalk, y tener métodos `ifTrue` e `ifFalse` que según el objeto ejecuten lo que le pasás o no.

$$\begin{array}{l}
\text{zero} \stackrel{\text{def}}{=} [\text{iszero} = \text{true}, \\
\quad \text{pred} = \varsigma(x)x, \\
\quad \text{succ} = \varsigma(x)(x.\text{iszero} := \text{false}).\text{pred} := x] \\
\text{uno} \stackrel{\text{def}}{=} \text{zero.succ} \\
\quad \rightarrow [\underbrace{\text{iszero} = \text{false}, \text{pred} = \text{zero}, \text{succ} = \dots}_{\text{uno}'}] \\
\text{dos} \stackrel{\text{def}}{=} \text{zero.succ.succ} \\
\quad \rightarrow [\text{iszero} = \text{false}, \text{pred} = \text{uno}', \text{succ} = \dots]
\end{array}$$

2.2.8 Codificando calculo λ

Podemos simular el cálculo λ no tipado,

$$M ::= M N \mid \lambda x.M \mid x$$

Queremos hacer esto porque los métodos del cálculo sigma no tienen argumentos, solo self. Idea intuitiva:

- Representar las funciones como objetos

$$[\text{arg} = \dots, \text{val} = \dots].$$

- Al aplicarlas, primero se asigna el valor del argumento al atributo *arg* y luego se envía el mensaje *val* que evalúa el cuerpo de la función
- De esa forma, una evaluación (o aplicación) $(f \ v)$ se traduce en $(o_f.\text{arg} := o_v).\text{val}$

Son esencialmente *method objects* de ing1

Vamos a definir una función $\llbracket \cdot \rrbracket : M \rightarrow a$ que dado un término nos da el objeto que lo codifica.

$$\begin{aligned} \llbracket x \rrbracket &\stackrel{\text{def}}{=} x \\ \llbracket M N \rrbracket &\stackrel{\text{def}}{=} (\llbracket M \rrbracket).\text{arg} := \llbracket N \rrbracket).\text{val} \\ \llbracket \lambda x.M \rrbracket &\stackrel{\text{def}}{=} [\text{val} = \varsigma(y)\llbracket M \rrbracket\{y.\text{arg}/x\}, \\ &\quad \text{arg} = \varsigma(y)y.\text{arg}] \\ &\quad \text{con } y \notin \text{fv}(M) \end{aligned}$$

en $\lambda x.M$ M tiene apariciones libres de x , entonces las reemplazamos por $y.\text{arg}$ para que funcione bien la semántica de la aplicación. En la codificación de la función, *arg* lo dejamos inicialmente como algo indefinido, porque no tiene sentido que tenga nada real asignado ya que siempre lo vamos a reemplazar.

Si hacemos $\llbracket \lambda x.x \rrbracket.\text{val}$, se cuelga.

Ejemplo 2.1. Ejemplos de codificación

$$\begin{aligned} \llbracket \lambda x.x \rrbracket &\stackrel{\text{def}}{=} [\text{val} = \varsigma(y)\llbracket x \rrbracket\{y.\text{arg}/x\}, \text{arg} = \varsigma(y)y.\text{arg}] \\ &= [\text{val} = \varsigma(y)x\{y.\text{arg}/x\}, \text{arg} = \varsigma(y)y.\text{arg}] \\ &= [\text{val} = \varsigma(y)y.\text{arg}, \text{arg} = \varsigma(y)y.\text{arg}] \\ \llbracket (\lambda x.x) M \rrbracket &\stackrel{\text{def}}{=} (\llbracket \lambda x.x \rrbracket.\text{arg} := \llbracket M \rrbracket).\text{val} \\ &= [\text{val} = \varsigma(y)y.\text{arg}, \text{arg} = \varsigma(y)y.\text{arg}].\text{arg} := \llbracket M \rrbracket.\text{val} \\ &\quad \longrightarrow \llbracket M \rrbracket \\ &\quad \text{siempre que } \llbracket M \rrbracket \text{ sea un objeto} \end{aligned}$$

Métodos con parámetros

Usando este truquito de codificación podemos extender a nuestros métodos para que reciban parámetros. Un método que espera un parámetro es un método cuya definición codifica a una función

$$\varsigma(y)\llbracket \lambda x.M \rrbracket$$

como notación, vamos a escribir

- $\lambda(x)M$ en vez de $\llbracket \lambda x.M \rrbracket$
- $M(N)$ en vez de $\llbracket M N \rrbracket$.

Ejemplo: PUnto en el plano

Lo definimos arrancando en el origen de coordenadas pero puede ser desplazado

$$\begin{aligned} \text{origen} &\stackrel{\text{def}}{=} [x = 0, \\ &\quad y = 0, \\ &\quad mv_x = \varsigma(p)\lambda(d_x)p.x := p.x + d_x, \\ &\quad mv_y = \varsigma(p)\lambda(d_y)p.y := p.y + d_y] \end{aligned}$$

$$\text{unidad} \stackrel{\text{def}}{=} \text{origen}.mv_x(1).mv_y(1)$$

2.2.9 Codificación de clases

Partimos de una def de objetos que tenia métodos. Le dimos semántica, agregamos un encoding (no cambió la semántica) de parámetros, y ahora vamos a hacer otro encoding para **generadores de objetos** como los que vimos en prototipado pero ahora clases

(Stateless) Trait

Un **trait** es una colección de ciertos métodos. Los stateless son un conjunto particular de los traits que no tienen estado: no especifican variables ni estado ni acceden al estado (i.e self). Una clase de construye a partir de traits, a veces se usan para interfaces.

Ejemplo 2.2. Ejemplo de trait

$$\begin{aligned} \text{CompT} &\stackrel{\text{def}}{=} [eq = \varsigma(t)\lambda(x)\lambda(y)(x.comp(y)) == 0, \\ &\quad lt = \varsigma(t)\lambda(x)\lambda(y)(x.comp(y)) < 0] \end{aligned}$$

Observar que en el cuerpo de los métodos eq y lt , no se usa t (self).

Los podemos pensar como una colección de **pre-métodos**: algo que eventualmente será un método

- Un pre método es $\varsigma(t)\lambda(y)b$ con $t \notin \text{fv}(\lambda(y)b)$ (i.e no usan self)
- En este caso por notación al ser un atributo podíamos omitir el $\varsigma(t)$ y escribir directamente $\lambda(y)b$, por lo que los traits pasarían a ser

$$\mathbf{t} = [l_i = \lambda(y_i)b_i \quad i \in 1..n]$$

A partir de un trait $\mathbf{t} = [l_i = \lambda(y_i)b_i \quad i \in 1..n]$ podemos definir un constructor de objetos (cuando t es completo, tiene todos los métodos que necesita)

$$\text{new} \stackrel{\text{def}}{=} \lambda(z)[l_i = \varsigma(s)z.l_i(s)^{i \in 1..n}]$$

$$\begin{aligned} o &\stackrel{\text{def}}{=} \text{new}(t) \\ &\approx [l_i = \varsigma(s)t.l_i(s)^{i \in 1..n}] \\ &\approx [l_i = \varsigma(y_i)b_i \quad i \in 1..n] \end{aligned}$$

new aprovecha el trait para crear un método real (uno en el que el self importa). Probablemente el y_i sea lo que querramos usar después como self.

Ejemplo. Ejemplo de *new*

$$\begin{aligned}
\text{CompT} &\stackrel{\text{def}}{=} [\text{eq} = \varsigma(t)\lambda(x)\lambda(y)(x.\text{comp}(y)) == 0, \\
&\quad \text{lt} = \varsigma(t)\lambda(x)\lambda(y)(x.\text{comp}(y)) < 0] \\
\text{new} &\stackrel{\text{def}}{=} \lambda(z)[l_i = \varsigma(s)z.l_i(s)^{i \in 1..n}] \\
\text{new}(\text{CompT}) &\approx [\text{eq} = \varsigma(s)\text{CompT}.\text{eq}(s), \\
&\quad \text{lt} = \varsigma(s)\text{CompT}.\text{lt}(s)] \\
&\approx [\text{eq} = \varsigma(x)\lambda(y)(x.\text{comp}(y)) == 0, \\
&\quad \text{lt} = \varsigma(x)\lambda(y)(x.\text{comp}(y)) < 0]
\end{aligned}$$

Acá el objeto creado es inutilizable, porque *CompT* usa *comp* que no es un método que define (no es completo).

Clases

Una **clase** va a ser un *trait* (completo) que además provea un método *new*.

$$\begin{aligned}
c &\stackrel{\text{def}}{=} [\text{new} = \varsigma(z)[l_i = \varsigma(s)z.l_i(s)^{i \in 1..n}], \\
&\quad l_i = \lambda(s)b_i^{i \in 1..n}]
\end{aligned}$$

Luego,

$$\begin{aligned}
o &\stackrel{\text{def}}{=} c.\text{new} \\
&\longrightarrow [l_i = \varsigma(s)c.l_i(s)^{i \in 1..n}] \\
&\approx [l_i = \varsigma(s)b_i^{i \in 1..n}]
\end{aligned}$$

Observar que las clases son traits que **no** son *stateless*, ya que *new* tiene *z* que hace referencia a self. Si no fuera así, para instanciar una clase nueva habría que hacer *c.new(c)* en vez de *c.new*, lo cual no tiene mucho sentido.

Ejemplo. Clase Contador

$$\begin{aligned}
\text{Contador} &\stackrel{\text{def}}{=} [\text{new} = \varsigma(z)[\quad v = \varsigma(s)z.v(s), \\
&\quad \text{inc} = \varsigma(s)z.\text{inc}(s), \\
&\quad \text{get} = \varsigma(s)z.\text{get}(s)], \\
&\quad v = \lambda(s)0, \\
&\quad \text{inc} = \lambda(s)s.v := s.v + 1, \\
&\quad \text{get} = \lambda(s)s.v]
\end{aligned}$$

Herencia

Si tenemos una clase

$$\begin{aligned}
c &\stackrel{\text{def}}{=} [\text{new} = \varsigma(z)[l_i = \varsigma(s)z.l_i(s)^{i \in 1..n}], \\
&\quad l_i = \lambda(s)b_i^{i \in 1..n}]
\end{aligned}$$

Queremos definir c' como subclase de c que agregue los pre-métodos $\lambda(s)b_k^{k \in n+1..n+m}$. Alcanza con agregar los métodos nuevos, y para los viejos referenciar a la superclase.

$$\begin{aligned} c &\stackrel{\text{def}}{=} [\text{new} = \varsigma(z)[l_i = \varsigma(s)z.l_i(s)^{i \in 1..n+m}], \\ &\quad l_j = c.l_j^{j \in 1..n}, \\ &\quad l_k = \lambda(s)b_k^{k \in n+1..n+m}] \end{aligned}$$

también si quisiéramos podríamos redefinir pre-métodos. En vez de delegar a la superclase, los definimos y ya.

Chapter 3

Paradigma lógico

Hasta ahora, vimos

- **Imperativo:** tenemos un estado compuesto por variables que se va modificando con las instrucciones, al final determinando un estado final.
- **Funcional:** tenemos expresiones que se van reduciendo hasta una forma que si todo sale bien es un valor
- **Objetos:** objetos como una forma de modelar la realidad, que colaboran mediante mensajes. Los métodos se implementan de forma imperativa, pero no es lo fundamental del paradigma. Eso es la colaboración mediante mensajes.

Se basa en el uso de la lógica como forma de programar. En vez de pensar en un algoritmo dado un problema, nos mantenemos en la misma "esfera". En vez de programar un problema, quedarnos en el *qué*.

Vamos a especificar **hechos** y **reglas de inferencia** y un **goal** (objetivo) a probar. Un motor de inferencia trata de probar que el goal es consecuencia de los hechos y reglas.

Es **declarativo**: se especifican hechos, reglas y goals sin indicar *cómo* se obtiene el último a partir de los primeros.

Vamos a usar **Prolog**:

- Los programas se escriben en un subconjunto de la logica de primer orden (cláusulas de Horn)
- El mecanismo teórico en el que se basa es el **método de resolución** (forma de dada una fórmula ver si es SAT o UNSAT)
- Para motivarlo, primero vamos a ver como es resolución en lógica proposicional

3.1 Resolución para Lógica Proposicional

3.1.1 Lógica proposicional

Sintaxis

Dado un conjunto \mathcal{V} de **variables proposicionales**, podemos definir inductivamente al conjunto de **fórmulas proposicionales** (o **proposiciones**) **Prop** de la siguiente manera,

1. Una variable proposicional P_0, P_1, \dots es una proposición
2. Si A, B son proposiciones, entonces también lo son
 - $\neg A$ (negación)

- $A \wedge B$ (conjunción)
- $A \vee B$ (disyunción)
- $A \supset B$ (implicación)
- $A \iff B$

Ejemplo. Ejemplos de fórmulas: $A \vee \neg B, A \wedge B \supset A \vee B$

Semántica

Una **valuación** es una función $v : \mathcal{V} \rightarrow \{T, F\}$ que asigna valores de verdad a las variables proposicionales. **Satisface** una proposición A si $v \models A$ donde,

$$\begin{aligned} v \models P &\text{ sii } v(P) = T \\ v \models \neg A &\text{ sii } v \not\models A \text{ (i.e no } v \models A) \\ v \models A \vee B &\text{ sii } v \models A \text{ o } v \models B \\ v \models A \wedge B &\text{ sii } v \models A \text{ y } v \models B \\ v \models A \supset B &\text{ sii } v \not\models A \text{ o } v \models B \\ v \models A &\iff B \text{ sii } (v \models A \text{ sii } v \models B) \end{aligned}$$

Tautologías y satisfacibilidad

Una proposición A es

- una **tautología** si $v \models A$ para toda valuación v (ej: $P \vee \neg P$)
- **satisfacible** si existe una valuación v tal que $v \models A$
- **insatisfacible** si no es satisfacible

Se puede extender a un conjunto. Un conjunto de proposiciones S es

- **satisfacible** si existe una valuación v tal que para todo $A \in S$, $v \models A$. (otra forma de verlo: hace verdadera a la conjunción de todas)
- **insatisfacible** si no es satisfacible

Ejemplo. Ejemplos de tautologías

- $A \supset A$
- $\neg\neg A \supset A$
- $(A \supset B) \iff (\neg B \supset \neg A)$ (contrarecíproco)

Ejemplo. Ejemplos de proposiciones insatisfacible

- $(\neg A \vee B) \wedge (\neg A \vee \neg B) \wedge A$
- $(A \supset B) \wedge A \wedge \neg B$

Una forma de probar tautologías es con una tabla de verdad. Sin importar los valores de las variables proposicionales, tiene que dar verdadero. Para probar que algo es insat, lo mismo pero viendo que todo da falso.

Teorema 3.1. Una proposición A es una tautología sii $\neg A$ es insatisfacible.

Dem. Por la ida y la vuelta

\Rightarrow) Si A es taut. para toda valuación v , $v \models A$. Entonces $v \not\models \neg A$.

\Leftarrow) Si $\neg A$ es insatisfacible, para toda valuación v $v \not\models \neg A$. Luego $v \models A$.

□

Esto sugiere un método indirecto para probar que una prop A es una tautología, probando que $\neg A$ es insatisfacible.

Suele ser más fácil refutar que probar. Entonces si queremos probar P , refutamos $\neg P$. Esto entonces va a ser útil porque a los mecanismos de resolución les va a ser más fácil refutar.

3.1.2 Resolución

Def (Principio de demostración por **refutación**). Vamos a probar que A es **válido** mostrando que $\neg A$ es **insatisfacible**.

Hay varias técnicas de demostración por refutación

- Tableaux semántico (1960)
- Procedimiento de Davis-Putnam (1960)
- **Resolución** (1965) (nos vamos a enfocar en este)

El método de resolución fue introducido por Alan Robinson en 1965, es simple de implementar, y se usa mucho en el ámbito de demostración automática de teoremas. Tiene una única regla de inferencia: **regla de resolución**. Y si bien no es imprescindible, por conveniencia vamos a asumir que las fórmulas están en **forma normal conjuntiva** (FNC).

Forma normal conjuntiva (FNC)

Un **literal** es una variable proposicional P o su negación $\neg P$. Una proposición A está en **FNC** si es una conjunción

$$C_1 \wedge \cdots \wedge C_n$$

donde cada C_i (llamado **cláusula**) es una disyunción

$$B_{i1} \vee \cdots \vee B_{in_i}$$

y cada B_{ij} es un literal. Una FNC es entonces una "conjunción de disyunciones de literales"

Ejemplo. Ejemplos de FNC

- $(P \vee Q) \wedge (P \vee \neg Q)$ está en FNC
- $(P \vee Q) \wedge (P \vee \neg \neg Q)$ no está en FNC que tiene un doble neg en C_2 .
- $(P \wedge Q) \vee P$ no está en FNC.

Teorema 3.2. Para toda proposición A puede hallarse una proposición A' en FNC que es lógicamente equivalente para A .

■ Decimos que A es lógicamente equivalente a B sii $A \iff B$ es una tautología.

Notación conjuntista de FNC

Tomando en cuenta que tanto el \vee como \wedge ,

- son conmutativos $((A \vee B) \iff (B \vee A))$
- son asociativos $((A \vee B) \vee C \iff A \vee (B \vee C))$

- son idempotentes $((A \vee A) \iff A)$

podemos asumir que

- Cada cláusula C_i es **distinta** (si fueran iguales, por idempotencia las podría juntar).
- Cada cláusula puede verse como un conjunto de literales distintos.

Por lo tanto podemos notar una FNC como

$$\{C_1, \dots, C_n\}$$

donde cada C_i es un conjunto de literales

$$\{B_{i1}, \dots, B_{in_i}\}$$

Ejemplo. La FNC $(P \vee Q) \wedge (P \vee \neg Q)$ se nota como

$$\{\{P, Q\}, \{P, \neg Q\}\}$$

Principio fundamental del método de resolución

La resolución se basa en el hecho de que la siguiente proposición es una **tautología**,

$$(A \vee P) \wedge (B \vee \neg P) \iff (A \vee P) \wedge (B \vee \neg P) \wedge (A \vee B)$$

Intuitivamente, si vale P entonces no puede valer $\neg P$, por lo que tiene que valer B valga o no A , y lo mismo al revés

Por lo tanto, el conjunto de cláusulas

$$\{C_1, \dots, C_m, \{A, P\}, \{B, \neg P\}\}$$

es **lógicamente equivalente** a

$$\{C_1, \dots, C_m, \{A, P\}, \{B, \neg P\}, \{A, B\}\}$$

y entonces el primero va a ser **insatisfacible** sii el segundo lo es.

La cláusula $\{A, B\}$ se llama **resolvente** de las cláusulas $\{A, P\}$ y $\{B, \neg P\}$.

El resolvente de las cláusulas $\{P\}$ y $\{\neg P\}$ es la **cláusula vacía** (porque es insatisfacible) y se anota \square . El vacío significa *falso*, como era un y de todo, podemos concluir que es insatisfacible.

Regla de resolución

Def (Opuesto). Dado un literal L , el **opuesto** de L \bar{L} se define como

- $\neg P$ si $L = P$
- P si $L = \neg P$

I Tenemos que definir esto para poder devolver la negación de un literal sin agregarle símbolos.

Def (Resolvente). Dadas dos cláusulas C_1, C_2 una cláusula C se dice **resolvente de C_1 y C_2** sii para algún literal $L, L \in C_1, \bar{L} \in C_2$ y

$$C = (C_1 - \{L\}) \cup (C_2 - \{\bar{L}\})$$

■ Esto es lo de antes pero escrito más formal.

Ejemplo. Ejemplos

- Las cláusulas $\{A, B\}$ y $\{\neg A, \neg B\}$ tienen dos resolventes: $\{A, \neg A\}$ y $\{B, \neg B\}$.
- Las cláusulas $\{P\}$ y $\{\neg P\}$ tienen a la cláusula vacía \square como resolvente.

Def (Regla de resolución). En base a lo anterior podemos definir la **regla de resolución** como sigue,

$$\frac{\{A_1, \dots, A_m, Q\} \quad \{B_1, \dots, B_n, \neg Q\}}{\{A_1, \dots, A_m, B_1, \dots, B_n\}}$$

Ejemplo. El resultado de aplicar la regla de resolución a

$$\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\}$$

es

$$\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}, \{P\}\}$$

Método de resolución

El proceso de agregar a un conjunto S el resolvente C de dos cláusulas $C_1, C_2 \in S$ (aplicando la regla de resolución a S) se llama **paso de resolución**. (asumimos $C \notin S$)

Recordamos que los pasos de resolución preservan la insatisfacibilidad,

$$S \text{ es insatisfacible} \iff S \cup \{C\} \text{ es insatisfacible}$$

ya que eran lógicamente equivalentes, como vimos en [sección 3.1.2 Principio fundamental del método de resolución](#)

Vamos a aplicar el metodo para obtener una **refutación**. Un conjunto de cláusulas se llama una refutación si contiene la cláusula vacía (\square).

De esa forma, el método de resolución trata de construir una secuencia de conjuntos de cláusulas obtenidas usando pasos de resolución hasta llegar a una **refutación**

$$S_1 \Rightarrow S_2 \Rightarrow \dots S_{n-1} \Rightarrow S_n \ni \square$$

en este caso, sabemos que el *conjunto inicial* de cláusulas es insatisfacible, dado que cada paso de resolución preserva la insatisfacibilidad y el último conjunto de cláusulas es insatisfacible.

Ejemplos

Ejemplo. $\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\}$ es insatisfacible.

1. $\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}\}$
2. $\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}, \{P\}\}$
3. $\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}, \{P\}, \{\neg P\}\}$
4. $\{\{P, Q\}, \{P, \neg Q\}, \{\neg P, Q\}, \{\neg P, \neg Q\}, \{P\}, \{\neg P\}, \square\}$

Ejemplo. $\{\{A, B, \neg C\}, \{A, B, C\}, \{A, \neg B\}, \{\neg A\}\}$ es insatisfacible

1. $\{\{A, B, \neg C\}, \{A, B, C\}, \{A, \neg B\}, \{\neg A\}\}$
2. $\{\{A, B, \neg C\}, \{A, B, C\}, \{A, \neg B\}, \{\neg A\}, \{A, B\}\}$
3. $\{\{A, B, \neg C\}, \{A, B, C\}, \{A, \neg B\}, \{A, B\}, \{\neg A\}, \{A\}\}$
4. $\{\{A, B, \neg C\}, \{A, B, C\}, \{A, \neg B\}, \{\neg A\}, \{A, B\}, \{A\}, \square\}$

Formulas satisfacibles

Queremos mostrar que $S = \{\{A, B, C\}, \{A\}, \{B\}\}$ es insatisfacible, pero no podemos aplicar ningún paso de resolución a S porque no hay negaciones. Por lo tanto, no puede llegarse a una refutación a partir de S . En lógica proposicional, estamos seguros entonces que S debe ser satisfacible. Efectivamente, podemos tomar por ej. $v(A) = v(B) = \mathbf{T}$.

Terminación

La aplicación reiterada de la regla de resolución **siempre termina** (suponiendo que el resolvente que se agrega es nuevo). Ya que,

1. El resolvente (la cláusula nueva que se agrega) se forma con los literales distintos que aparecen en el conjunto de las cláusulas de partida S .
2. Hay una cantidad **finita** de literales en el conjunto de cláusulas de partida S .

En el peor de los casos, la regla de resolución podrá generar una nueva cláusula por cada combinación diferente de literales distintos de S . Pero sigue siendo finita y por lo tanto termina.

Corrección y completitud

Teorema 3.3. Dado un conjunto finito S de cláusulas, S es insatisfacible sii tiene una refutación.

Este resultado establece la corrección y completitud del método de resolución.

En LPO no es tan bueno. Si encuentra una refutación significa que es insatisfacible pero puede pasar que no la encontremos

Resumen del algoritmo

Para probar que A es una tautología hacemos lo siguiente,

1. Calculamos la FNC de $\neg A$
2. Aplicamos el método de resolución
3. Si hallamos una refutación, entonces $\neg A$ es insatisfacible y por lo tanto A es una tautología. (Por Teo. 3.1)
4. Si no hallamos ninguna refutación, entonces $\neg A$ es satisfacible y por lo tanto A no es una tautología. (Por negación de Teo. 3.1)

3.2 Lógica de Primer Orden

3.2.1 Sintaxis

Def (Lenguaje de primer orden). Un lenguaje de primer orden (LPO) \mathcal{L} consiste en

1. Un conjunto numerable de **constantes** c_0, c_1, \dots
2. Un conjunto numerable de **símbolos de función** con aridad $n > 0$ (indicando el número de argumentos) f_0, f_1, \dots
3. Un conjunto numerable de **símbolos de predicado** con aridad $n \geq 0$, P_0, P_1, \dots . Si $n = 0$, es una variable proposicional.

Def (Término). Sea $\mathcal{V} = \{x_0, x_1, \dots\}$ un conjunto numerable de variables y \mathcal{L} un LPO. El conjunto de **\mathcal{L} -términos** se define inductivamente como

1. Toda constante de \mathcal{L} y toda variable es un \mathcal{L} -término.

2. Si $t_1, \dots, t_n \in \mathcal{L}$ -términos y f es un símbolo de función de aridad n , entonces $f(t_1, \dots, t_n) \in \mathcal{L}$ -términos

Def (Fórmulas atómicas). Sea \mathcal{V} un conjunto numerable de variables y \mathcal{L} un LPO. El conjunto de \mathcal{L} -fórmulas atómicas se define inductivamente como,

1. Todo símbolo de predicado de aridad 0 (i.e vars proposicionales) es una \mathcal{L} -fórmula atómica
2. Si $t_1, \dots, t_n \in \mathcal{L}$ -términos y P es un símbolo de predicado de aridad n , entonces $P(t_1, \dots, t_n)$ es una \mathcal{L} -fórmula atómica.

Def (Fórmulas de primer orden). Sea \mathcal{V} un conjunto numerable de variables y \mathcal{L} un LPO. El conjunto de \mathcal{L} -fórmulas se define inductivamente como,

1. Toda \mathcal{L} -fórmula atómica es una \mathcal{L} -fórmula
2. Si $A, B \in \mathcal{L}$ -fórmulas, entonces las siguientes son \mathcal{L} -fórmulas
 - $A \vee B$
 - $A \wedge B$
 - $A \supset B$
 - $A \iff B$
 - $\neg A$.
3. Para toda variable x_i y cualquier \mathcal{L} -fórmula A , $\forall x_i. A$ y $\exists x_i. A$ son \mathcal{L} -fórmulas

Ejemplo. LPO para aritmética

- Constantes: 0
- Símbolos de función: $S, +, *$
- Símbolos de predicado: $<, =$
- Términos: $S(0), +(S(0), S(S(0))), *(S(x_1), +(x_2, S(x_3)))$
- Fórmulas atómicas: $<(0, S(0)), <(x_1, +(S(0), x_2))$
- Fórmulas:
 - $\forall x. \forall y. (x < y \supset \exists z. y = x + z)$
 - $\forall x. \forall y. ((x < y \wedge y < x) \vee x = y)$

Def (Variables libres y ligadas). Las variables pueden ocurrir **libres** o **ligadas**. Los cuantificadores ligan a las variables, y usamos $FV(A)$ para referirnos a las libres y $BV(A)$ para las ligadas. Ambos se pueden definir por inducción estructural en A .

Ejemplo. Si $A = \forall x. (R(x, y) \supset P(x))$, entonces $FV(A) = \{y\}$ y $BV(A) = \{x\}$.

Una fórmula A se dice **rectificada** si $FV(A)$ y $BV(A)$ son disjuntos y cuantificadores distintos de A ligan a variables distintas. Toda fórmula se puede rectificar (renombrando vars ligadas) a una lógicamente equivalente.

Esto nos sirve para evitar problemas y que sea todo lindo. Por ej. si tenemos $\forall x. \forall x. A$ es confuso, pero lo podemos renombrar para que sea $\forall x. \forall y. A$

Def (Sentencia). Una **sentencia** es una fórmula cerrada (sin variables libres).

Muchos resultados se formulan para sentencias, lo cual no implica una pérdida de generalidad ya que toda fórmula es lógicamente equivalente a su **clausura universal** (la fórmula que tiene adelante \forall para cada variable libre)

$$\forall x. \forall y. P(x, y) \iff P(x, y)$$

3.2.2 Semántica

Def (Estructura). Dado un LPO \mathcal{L} una **estructura para \mathcal{L}** , M , es un par $M = (M, I)$ en donde

- M (**dominio**) es un conjunto no vacío
- I (**función de interpretación**) asigna funciones y predicados sobre M a símbolos de \mathcal{L} de la siguiente manera,
 1. Para toda constante c , $I(c) \in M$
 2. Para todo f de aridad $n > 0$, $I(f) : M^n \rightarrow M$
 3. Para todo predicado P de aridad $n \geq 0$, $I(P) : M^n \rightarrow \{T, F\}$

Def (Asignación). Sea M una estructura para \mathcal{L} . Una **asignación** es una función $s : \mathcal{V} \rightarrow M$.

Dado $a \in M$ usamos la notación $s[x \leftarrow a]$ para denotar la asignación que se comporta igual a s salvo en elemento x , para el cual retorna a .

Def (Satisfacibilidad). La relación $s \models_M A$ establece que la asignación s satisface la fórmula en la estructura M . La definimos de manera informal por inducción estructural en A .

$$\begin{array}{ll}
 s \models_M P(t_1, \dots, t_n) & \text{sii } P_M(s(t_1), \dots, s(t_n)) \\
 s \models_M \neg A & \text{sii } s \not\models_M A \\
 s \models_M (A \wedge B) & \text{sii } s \models_M A \text{ y } s \models_M B \\
 s \models_M (A \vee B) & \text{sii } s \models_M A \text{ o } s \models_M B \\
 s \models_M (A \supset B) & \text{sii } s \not\models_M A \text{ o } s \models_M B \\
 s \models_M (A \iff B) & \text{sii } (s \models_M A \text{ sii } s \models_M B) \\
 s \models_M \forall x_i. A & \text{sii } s[x_i \leftarrow a] \models_M A \text{ para todo } a \in M \\
 s \models_M \exists x_i. A & \text{sii } s[x_i \leftarrow a] \models_M A \text{ para algún } a \in M
 \end{array}$$

Def (Validez). Decimos que una fórmula A es,

- **satisfactible en M** sii existe alguna asignación s tal que $s \models_M A$.
- **satisfactible** sii existe un M tal que A es satisfactible en M . En caso contrario, decimos que A es **insatisfactible**.
- **válida en M** sii $s \models_M A$ para toda asignación s
- **válida** sii es válida en toda estructura M .

Y A es válida sii $\neg A$ es insatisfactible.

Teorema 3.4 (Teorema de Church). **No** existe un algoritmo que pueda determinar si una fórmula de primer orden es válida.

Como consecuencia del Teorema de Church, el método de resolución de primer orden no va a ser un **procedimiento efectivo** (como el de proposicional), sino que va a ser uno de **semi decisión**:

- Si una sentencia es insatisfactible hallará una refutación,
- Pero si es satisfactible puede que no se detenga.

3.3 Resolución en lógica de primer orden

Así como el lenguaje es más complejo, vamos a tener que hacer algunas cosas más para hacer lo mismo que en proposicional.

3.3.1 Forma clausal

Es una FNC en notación de conjuntos, análogo a la forma clausal de proposicional, pero requiere tener en cuenta los **cuantificadores** (\forall, \exists). El pasaje consiste en 6 pasos de conversión,

1. Escribir la fórmula en términos de $\vee, \wedge, \neg, \forall, \exists$ (**eliminar implicación**).

Para esto se usa $(P \supset A) \iff (\neg P \vee Q)$

2. Pasar a **forma normal negada**.
3. Pasar a **forma normal prenexa** (opcional).
4. Pasar a **forma normal de Skolem**. (*esta es la única picante*)
5. Pasar matriz a **forma normal conjuntiva**.
6. **Distribuir** los cuantificadores universales.

Todos los pasos preservan **validez lógica**, salvo la skolemización que preserva la **satisfactibilidad**.

Forma normal negada

Movemos las negaciones lo más adentro posible. El conjunto de fórmulas en **forma normal negada** (FNN) se define inductivamente como,

1. Para cada fórmula atómica A , A y $\neg A$ están en FNN.
2. Si $A, B \in \text{FNN}$, entonces $A \vee B, A \wedge B \in \text{FNN}$
3. Si $A \in \text{FNN}$, entonces $\forall x.A, \exists x.A \in \text{FNN}$

Prop. Toda fórmula es **lógicamente equivalente** a otra en FNN.

Dem. Por inducción estructural usando las siguientes equivalencias,

$$\neg A \wedge B \iff \neg A \vee \neg B$$

$$\neg A \vee B \iff \neg A \wedge \neg B$$

$$\neg \neg A \iff A$$

$$\neg \forall x.A \iff \exists x.\neg A$$

$$\neg \exists x.A \iff \forall x.\neg A$$

□

Ejemplo. Ejemplos,

- $\neg \exists x.((\neg P(x) \vee \exists y.R(x, y)) \supset (\exists z.R(x, z) \vee P(a)))$ no está en FNN. Se transforma en,
- $\forall x.((P(x) \vee \exists y.R(x, y)) \wedge (\forall z.\neg R(x, z) \vee \neg P(a)))$

Forma normal prenexa

■ Escribir la fórmula de forma tal que luzca *ordenada*, con los cuantificadores al principio.

Es una fórmula de la forma $Q_1 x_1 \dots Q_n x_n. B, n \geq 0$ donde

- B no tiene cuantificadores (llamada **matriz**)
- x_1, \dots, x_n son variables.
- $Q_i \in \{\forall, \exists\}$ son cuantificadores.

Prop. Toda fórmula rectificadora (con variables renombradas adecuadamente) es lógicamente equivalente a una fórmula B en forma prenexa.

Proof. Por inducción estructural, asumiendo que A está en FNN y usando,

$$\begin{array}{ll} (\forall x.A) \wedge B \iff \forall x.(A \wedge B) & (\forall x.A) \vee B \iff \forall x.(A \vee B) \\ (A \wedge \forall x.B) \iff \forall x.(A \wedge B) & (A \vee \forall x.B) \iff \forall x.(A \vee B) \\ (\exists x.A) \wedge B \iff \exists x.(A \wedge B) & (\exists x.A) \vee B \iff \exists x.(A \vee B) \\ (A \wedge \exists x.B) \iff \exists x.(A \wedge B) & (A \vee \exists x.B) \iff \exists x.(A \vee B) \end{array}$$

□

Ejemplo. Ejemplo

1. $\forall x. \neg P(x) \wedge (\exists y. Q(y) \vee \forall z. P(z))$ (regla 3.b)
2. $\forall x. \neg P(x) \wedge (\exists y. (Q(y) \vee \forall z. P(z)))$ (regla 4)
3. $\exists y. (\forall x. \neg P(x) \wedge (Q(y) \vee \forall z. P(z)))$ (regla 2.b)
4. $\exists y. (\forall x. \neg P(x) \wedge \forall z. (Q(y) \vee P(z)))$ (regla 2.a)
5. $\exists y. \forall z. (\forall x. \neg P(x) \wedge (Q(y) \vee P(z)))$ (regla 1.a)
6. $\exists y. \forall z. \forall x. (\neg P(x) \wedge (Q(y) \vee P(z)))$ está en forma prenexa.

Forma normal de Skolem

Hasta ahora tenemos una fórmula que

1. Está escrita en términos de $\wedge, \vee, \neg, \forall, \exists$.
2. Si tiene negaciones, solo se aplican a átomos (**forma normal negada**)
3. (opcionalmente) Si tiene cuantificadores, se encuentran todos en el prefijo (**forma normal prenexa**)

El proceso de pasar una fórmula a forma normal de Skolem se llama **skolemización**, e involucra eliminar los cuantificadores existenciales **sin** alterar *satisfactibilidad*.

Es probable que no podamos mantener que sean lógicamente equivalentes, pero que mantengan satisfactibilidad.

Cómo podemos cambiar los \exists sin cambiar la satisfactibilidad? Vamos a introducir *testigos* para cada uno.

Todo cuantificador existencial se va a instanciar en una constante o **función de skolem**. Estas se suelen conocer como **parámetros**.

Por ejemplo, $\exists x. P(x)$ se *skolemiza* a $P(c)$ donde c es una nueva constante que se agrega al lenguaje de primer orden. Esto preserva la satisfactibilidad, ya que si la primera era satisfactible, es porque existía x que cumpla $P(x)$. Por lo tanto, voy a poder armar un modelo M que asigne en c ese valor, y $P(c)$ también va a ser satisfactible.

Gran frase de Diego: “En términos teóricos hay una interpretación que lo hace valer. Cuál es? Buscala, papá. Pero digo, existe.”

La **skolemización** es un mecanismo para eliminar existenciales.

Prop. Si A' es el resultado de skolemizar A , entonces A es satisfactible sii A' es satisfactible.

Como consecuencia, la skolemización preserva **insatisfactibilidad**, lo cuál será suficiente para aplicar el método de resolución.

No podemos eliminar existenciales, usando skolemización, sin alterar la validez. Por ejemplo, $\exists x. (P(a) \supset P(x))$ válida (satisfacible en toda estructura) mientras que $P(a) \supset P(b)$ no (podemos encontrar estructuras tal que $P(a)$ sea verdadero y $P(b)$ falso).

Def (Skolemización). Cada ocurrencia de una subfórmula $\exists x.B$ en A se reemplaza por

$$B\{x \leftarrow f(x_1, \dots, x_n)\}$$

donde

- $\bullet\{\bullet \leftarrow \bullet\}$ es la operación usual de sustitución (sustituir las ocurrencias libres de una variable en una expresión - fórmula o término - por otra expresión).
- f es un símbolo de función nuevo y x_1, \dots, x_n son las variables de las que depende x en B .
- Si $\exists x.B$ forma parte de una fórmula mayor, x solo depende de las **variables libres** de B . (por ej. en $\forall z.\forall y.\exists x.P(y, x)$ la var x depende de y)

Tengo que reemplazarlo por una función que depende de las variables porque para cada combinación de variables voy a tener un testigo distinto. Y reemplaza por una constante si no depende de ninguna variable.

Sea A una sentencia rectificada en FNN (no hace falta en prenexa). Una **forma normal de Skolem de A**, escrito $\mathbf{SK}(A)$, es una formula sin existenciales se obtiene recursivamente de la siguiente manera:

Sea A' cualquier subfórmula de A .

- Si A' es una fórmula atómica o su negación, $\mathbf{SK}(A') = A'$
- Si A' es de la forma $(B \star C)$ con $\star \in \{\wedge, \vee\}$, entonces $\mathbf{SK}(A') = (\mathbf{SK}(B) \star \mathbf{SK}(C))$.
- Si A' es de la forma $\forall x.B$, entonces $\mathbf{SK}(A') = \forall x.\mathbf{SK}(B)$.
- Si A' es de la forma $\exists x.B$ y $\{x, y_1, \dots, y_m\}$ son las variables libres de B (que se ligan en A , porque es sentencia), entonces

1. Si $m > 0$, crear un nuevo **símbolo de función de Skolem** f_x de aridad m y definir

$$\mathbf{SK}(A') = \mathbf{SK}(B\{x \leftarrow f_x(y_1, \dots, y_m)\})$$

2. Si $m = 0$ crear una nueva **constante de Skolem** c_x y

$$\mathbf{SK}(A') = \mathbf{SK}(B\{x \leftarrow c_x\})$$

Como A está rectificadada, cada f_x y c_x es única.

Ejemplo. Ejemplo de skolemización La forma normal de Skolem de la siguiente fórmula

$$\forall x. \left(P(a) \vee \exists y. (Q(y) \wedge \forall z. (P(y, z) \vee \exists u. Q(x, u))) \right) \vee \exists w. Q(w)$$

es

$$\forall x. \left(P(a) \vee Q(f_y(x)) \wedge \forall z. (P(f_y(x), z) \vee Q(x, f_u(x))) \right) \vee Q(c_w)$$

El algoritmo de la skolemización no es determinístico, te dice que agarres una subfórmula cualquiera y apliques la regla. Entonces por ejemplo para la sentencia

$$\forall x. \exists y. \exists z. R(x, y, z)$$

1. Alternativa 1 (rojo y después azul)

- (a) $\forall x. \exists y. \exists z. R(x, y, z)$
- (b) $\forall x. \exists z. R(x, f_y(x), z)$
- (c) $\forall x. R(x, f_y(x), f_z(x))$

2. Alternativa 2 (azul y después rojo)

- (a) $\forall x. \exists y. \exists z. R(x, y, z)$
- (b) $\forall x. \exists y. R(x, y, f_z(x, y))$
- (c) $\forall x. R(x, f_y(x), f_z(x, y))$

es por esto que nos conviene skolemizar de afuera hacia adentro, para que en cada paso tengamos más variables ligadas y por lo tanto menos parámetros.

Forma clausal

Hasta ahora tenemos una fórmula que

1. Está escrita en términos de $\wedge, \vee, \neg, \forall, \exists$.
2. Si tiene negaciones, solo se aplican a átomos (**forma normal negada**)
3. Si tiene cuantificadores, son universales (**forma normal de Skolem**)
4. Si está en **forma normal prenexa** y tiene cuantificadores, estos están en el prefijo

Tiene la pinta

$$\forall x_1 \dots \forall x_n. B$$

Dada la fórmula $\forall x_1 \dots \forall x_n. B$, vamos a

1. Pasar B (la **matriz**) a su forma normal conjuntiva B' como si fuera una fórmula proposicional (ver **section 3.1.2 Forma normal conjuntiva (FNC)**, es una conjunción de disyunciones de literales)
2. Distribuir los cuantificadores sobre cada conjunción usando la fórmula válida

$$\forall x. (A \wedge B) \iff \forall x. A \wedge \forall x. B$$

lo que nos da una conjunción de **cláusulas**

$$\forall x_1 \dots \forall x_n. C_1 \wedge \dots \wedge \forall x_1 \dots \forall x_n. C_m$$

donde cada C_i es una disyunción de literales.

3. Y lo podemos simplificar escribiendo $\{C_1, \dots, C_m\}$

Ejemplo. Ejemplo de pasaje a forma clausal (es la misma fórmula que antes pero en FNP)

$$\forall x. \forall z. (P(a) \vee (Q(g(x)) \wedge (P(g(x), z) \vee Q(x, f(x))))) \vee Q(c)$$

1. Pasamos a FNC

$$\forall x. \forall z. ([P(a) \vee Q(g(x)) \vee Q(c)] \wedge [P(a) \vee P(g(x)) \vee Q(x, f(x)) \vee Q(c)])$$

2. Distribuimos los cuantificadores

$$\begin{aligned} &\forall x. \forall z. [P(a) \vee Q(g(x)) \vee Q(c)] \wedge \\ &\forall x. \forall z. [P(a) \vee P(g(x)) \vee Q(x, f(x)) \vee Q(c)] \end{aligned}$$

3. Pasamos a notación de conjuntos

$$\begin{aligned} &\left\{ \{P(a), Q(g(x)), Q(c)\} \right. \\ &\quad \left. \{P(a), P(g(x)), Q(x, f(x)), Q(c)\} \right\} \end{aligned}$$

3.3.2 Regla de resolución

En proposicional teníamos la siguiente regla de resolución,

$$\frac{\{A_1, \dots, A_m, Q\} \quad \{B_1, \dots, B_n, \neg Q\}}{\{A_1, \dots, A_m, B_1, \dots, B_n\}}$$

Pero la podemos usar **directamente** para LPO?

Por ejemplo, si tenemos la siguiente fórmula no satisfactible

$$(\forall x. P(x)) \vee \neg P(a)$$

su forma clausal es

$$\{\{P(x)\}, \{\neg P(a)\}\}$$

no podemos aplicar la regla directamente porque $P(x) \neq P(a)$ (no son idénticos), pero son **unificables**! Queremos encontrar una sustitución de variables que haga que los términos queden iguales.

Ver [subsection 1.4.6 Unificación](#).

La regla de resolución para LPO va a ser,

$$\frac{\{B_1, \dots, B_k, A_1, \dots, A_m\} \quad \{\neg D_1, \dots, \neg D_j, C_1, \dots, C_n\}}{\sigma(\{A_1, \dots, A_m, C_1, \dots, C_n\})}$$

donde σ es el MGU de

$$\{B_1 \doteq B_2, \dots, B_{k-1} \doteq B_k, B_k \doteq D_1, \dots, D_{j-1} \doteq D_j\}$$

Quiero que sean varios porque tal vez tengo varias instanciaciones del predicado, tengo $P(x), P(y), P(z)$ y quiero que todos unifiquen a lo mismo.

- Vamos a asumir que las cláusulas $\{B_1, \dots, B_k, A_1, \dots, A_m\}$ y $\{\neg D_1, \dots, \neg D_j, C_1, \dots, C_n\}$ no tienen variables en común (c.c. se renombran).
- Obs que $\sigma(B_1) = \dots = \sigma(B_k) = \sigma(D_1) = \dots = \sigma(D_j)$
- La cláusula $\sigma(\{A_1, \dots, A_m, C_1, \dots, C_n\})$ se llama **resolvente** (de $\{B_1, \dots, B_k, A_1, \dots, A_m\}$ y $\{\neg D_1, \dots, \neg D_j, C_1, \dots, C_n\}$)

3.3.3 Método de resolución

Las siguientes nociones son análogas al caso de proposicional: Cláusula vacía, paso de resolución y refutación.

Teorema (Herbrand-Skolem-Gödel). Cada paso de resolución preserva la **satisfactibilidad**.

Ejemplo

Supongamos que dado A , obtenemos $\neg A$, lo convertimos a forma clausal y nos queda $C_1 \wedge C_2 \wedge C_3$ donde

1. $C_1 = \forall z_1. \forall x. (\neg P(z_1, a) \vee \neg P(z_1,) \vee \neg P(x, z_1))$
2. $C_2 = \forall z_2. (P(z_2, f(z_2)) \vee P(z_2, a))$
3. $C_3 = \forall z_3. (P(f(z_3), z_3) \vee P(z_3, a))$

En notación conjuntista,

$$\left\{ \begin{array}{l} \{\neg P(z_1, a), \neg P(z_1,), \neg P(x, z_1)\}, \\ \{P(z_2, f(z_2)), P(z_2, a)\}, \\ \{P(f(z_3), z_3), P(z_3, a)\} \end{array} \right\}$$

Hacemos la resolución,

$$\begin{aligned} C_1 &= \{\neg P(z_1, a), \neg P(z_1,), \neg P(x, z_1)\}, \\ C_2 &= \{P(z_2, f(z_2)), P(z_2, a)\}, \\ C_3 &= \{P(f(z_3), z_3), P(z_3, a)\} \end{aligned}$$

1. De C_1 y C_2 con $\{z_1 \leftarrow a, x \leftarrow a, z_2 \leftarrow a\}$

$$C_4 = \{P(a, f(a))\}$$

2. De C_1 y C_3 con $\{z_1 \leftarrow a, x \leftarrow a, z_3 \leftarrow a\}$

$$C_5 = \{P(f(a), a)\}$$

3. De C_1 y C_5 con $\{z_1 \leftarrow f(a), x \leftarrow a\}$

$$C_6 = \neg P(a, f(a))$$

4. De C_4 y C_6 : \square

3.3.4 Diferencias con proposicional

- Proposicional

$$\frac{\{A_1, \dots, A_m, Q\} \quad \{B_1, \dots, B_n, \neg Q\}}{\{A_1, \dots, A_m, B_1, \dots, B_n\}}$$

- Primer orden

$$\frac{\{B_1, \dots, B_k, A_1, \dots, A_m\} \quad \{\neg D_1, \dots, \neg D_j, C_1, \dots, C_n\}}{\sigma(\{A_1, \dots, A_m, C_1, \dots, C_n\})}$$

donde σ es el MGU de $\{B_1, \dots, b_k, D_1, \dots, D_j\}$

Regla de resolución binaria

Pero podríamos tener en LPO algo parecido, en el que solo se unifique de a dos?

$$\frac{\{B, A_1, \dots, A_m\} \quad \{\neg D, C_1, \dots, C_n\}}{\sigma(\{A_1, \dots, A_m, C_1, \dots, C_n\})}$$

donde σ es el MGU de $\{B, D\}$.

No, porque sería **incompleta**.

Ejemplo. Intentemos de refutar con la regla binaria $\{\{P(x), P(y)\}, \{\neg P(v), \neg P(w)\}\}$

1. $\{P(x), P(y)\}$
2. $\{\neg P(v), \neg P(w_1)\}$
3. (1 y 2) $\{P(y), \neg P(w_2)\}$ con $\sigma = \{x \leftarrow v\}$

4. $(3 \text{ y } 2) \{ \neg P(w_1), \neg P(w_2) \}$ con $\sigma\{y \leftarrow v\}$
5. ...

Sin importar cuales resolvamos, siempre nos quedan 2 cláusulas y nunca termina.

Notar que la razón por la cual se hace el renombre de w a w_1 y w_2 es porque por la definición de Forma Clausal para primer orden, ([section 3.3.1 Forma clausal](#)) cada cláusula está cuantificada universalmente por separado. De esa forma se pueden hacer los renombres para que no haya conflictos.

Acá me adelanto, un ejemplo podría ser si tenemos dos definiciones en prolog

```
p(X) :- q(X).
r(X) :- s(X).
% se puede reescribir como
p(X) :- q(X).
r(Y) :- s(Y).
```

no es la misma X en ambas cláusulas, sino que se trata de una variable diferente que puede unificar a cosas diferentes.

Una forma de arreglarlo es con una regla adicional de **factorización** que une los términos

$$\frac{\{B_1, \dots, B_k, A_1, \dots, A_m\}}{\sigma(\{B_1, A_1, \dots, A_m\})}$$

donde σ es el MGU de $\{B_1, \dots, B_k\}$

En el ejemplo anterior,

1. $\{\{P(x), P(y)\}, \{\neg P(v), \neg P(w)\}\}$
2. $\{\{P(x), P(y)\}, \{\neg P(v), \neg P(w)\}, \{P(z)\}\}$ (fact.)
3. $\{\{P(x), P(y)\}, \{\neg P(v), \neg P(w)\}, \{P(z)\}, \{\neg P(u)\}, \}$ (fact.)
4. $\{\{P(x), P(y)\}, \{\neg P(v), \neg P(w)\}, \{P(z)\}, \{\neg P(u)\}, \square\}$ (resolución binaria)

3.4 Resolución SLD

3.4.1 Resolución lineal

Hallar la resolución puede ser complicado, hay muchos órdenes posibles y es no determinístico. El espacio de búsqueda puede ser muy grande.

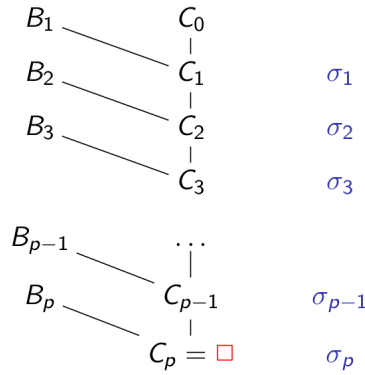
Hay mucho **no determinismo**:

- Qué cláusulas elegimos? **Regla de búsqueda**
- Qué literales eliminamos? **Regla de selección**

Son necesarias restricciones (ambas reglas) para reducir el espacio de búsqueda y que sea útil en la práctica. Y es deseable que estas no renuncien a la **completitud** del método.

Vamos a ver un método de resolución que cambia la estrategia de búsqueda.

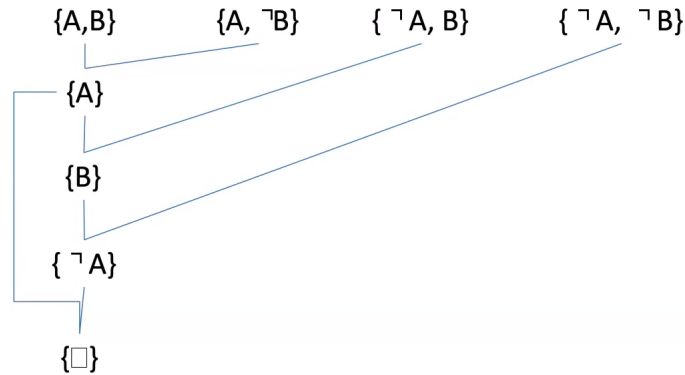
Def (Resolución lineal). Decimos que una secuencia de pasos a partir de S es **lineal** si es de la forma



donde C_0 y cada B_i es un elemento de S (o algún C_j con $j < i$)

En cada paso agarramos el último resolvente y lo intentamos de resolver contra alguno del conjunto de cláusulas o alguno de los resolventes anteriores.

Ejemplo. Ejemplo de resolución lineal



Características

- En general **reduce** el espacio de búsqueda considerablemente (restringimos que cláusulas podemos elegir)
- Preserva **completitud** (si la fórmula es insatisfactible, existe una resol lineal que llega a una refutación).
- Pero sigue siendo muy **no determinístico**. El criterio de búsqueda deja espacio para refinamientos y no se especificó ningún criterio de selección.

3.4.2 Cláusulas de Horn

Es un mecanismo que nos va a dar mayor eficiencia en el proceso de producir refutaciones, sin perder completitud, pero usando *subclases* de fórmulas. Una subclase es las **cláusulas de Horn**, que son disyunciones de literales que tienen **a lo sumo** un literal positivo.

Las formas clausales ([section 3.3.1 Forma clausal](#)) tenían la siguiente pinta

$$\forall x_{11} \dots \forall x_{1m_1}. C_1 \wedge \dots \wedge \forall x_{k1} \dots \forall x_{km_k}. C_k$$

una conjunción de **cláusulas** $\forall x_1 \dots \forall x_m C$ donde cad C es una disyunción de literales, y se escribían como $\{C'_1, \dots, C'_k\}$ donde cada C'_i es el conjunto de literales en C_i .

Def (Cláusula de Horn). Las **cláusulas de Horn** son de la forma $\forall x_1, \dots, \forall x_m C$ tal que la disyunción de literales C tiene **a lo sumo** un literal positivo.

C puede tomar una de las formas,

- $\{B, \neg A_1, \dots, \neg A_n\}$ (cláusula de definición)
- $\{B\}$ (verdades o hechos)
- $\{\neg A_1, \dots, \neg A_n\}$ (cláusula **goal**, objetivo o negativa)

No toda fórmula de primer orden puede expresarse como una cláusula de Horn, por ejemplo,

- $\forall x.(P(x) \vee Q(x))$ (tiene dos literales positivos)
- $\forall x.P(x) \vee \forall x.Q(x) \supset \forall x.(P(x) \vee Q(x))$

Sin embargo, es suficientemente expresivo como para representar programas en la visión de resolución como cómputo (lo vemos en Prolog).

Tipos de cláusulas

Una **cláusula de definición** (definite clause) es una de la forma $\forall x_1 \dots \forall x_m C$ tal que la disyunción de literales C tiene *exactamente* un literal positivo.

Y sea $S = P \cup \{G\}$ un conjunto de cláusulas de Horn (con nombres de vars disjuntos) tal que P es un conjunto de cláusulas de definición y G una negativa, decimos que son las **cláusulas de entrada**.

Donde,

- P se conoce como el **programa o base de conocimientos**, y
- G es el **goal, meta o cláusula objetivo**

3.4.3 Resolución SLD

| S: Selectiva L: Lineal D: Definición (Selective Linear Definite)

Una secuencia de pasos de **resolución SLD** para S es una secuencia

$$\langle N_0, N_1, \dots, N_p \rangle$$

de **cláusulas negativas** que satisfacen dos condiciones,

1. N_0 es el goal G
2. Para todo N_i en la secuencia, $0 < i < p$, si N_i es

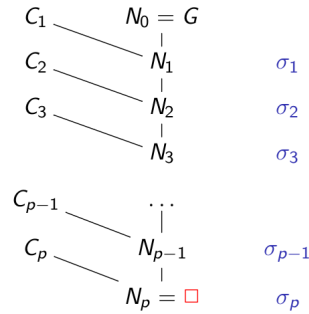
$$\{\neg A_1, \dots, \neg A_{k-1}, \neg A_k, \neg A_{k+1}, \dots, \neg A_n\}$$

entonces hay alguna **cláusula de definición** C_i de la forma $\{A, \neg B_1, \dots, \neg B_m\}$ en P tal que A_k y A son unificables con un MGU σ , y si

- $m = 0$ (i.e es una *verdad*) entonces N_{i+1} es
 $\sigma(\{\neg A_1, \dots, \neg A_{k-1}, \neg A_{k+1}, \dots, \neg A_n\})$
- $m > 0$ entonces N_{i+1} es
 $\sigma(\{\neg A_1, \dots, \neg A_{k-1}, \neg B_1, \dots, \neg B_m, \neg A_{k+1}, \dots, \neg A_n\})$

3.4.4 Refutación SLD

y entonces una **refutación SLD** es una secuencia de pasos de resolución SLD $\langle N_0, \dots, N_p \rangle$ tal que $N_p = \square$ (llega a una cláusula vacía)



De esa forma,

- En cada paso, se resuelven las cláusulas $\{\neg A_1, \dots, \neg A_{k-1}, \neg A_k, \neg A_{k+1}, \dots, \neg A_n\}$ (resolvente anterior) y $\{A, \neg B_1, \dots, \neg B_m\}$ (definición)
- Los átomos A_k y A se unifican con un MGU σ_i
- El literal A_k se llama **átomo seleccionado** de N_i
- Al final, tenemos la **sustitución respuesta**

$$\sigma_p \circ \dots \circ \sigma_1$$

(esta se usa en Prolog para extraer la salida del programa)

Ejemplo

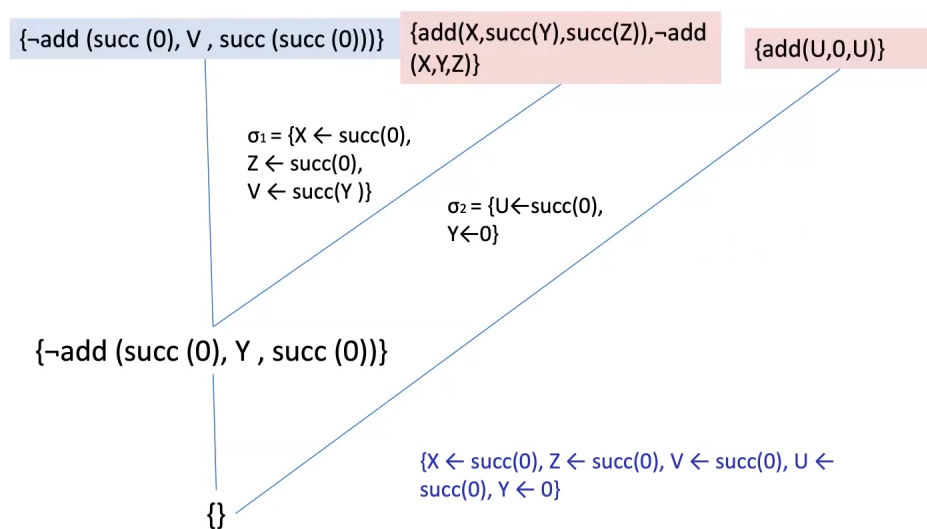
Consideremos las siguientes cláusulas de definición,

- $C_1 = \{add(U, 0, U)\}$
- $C_2 = \{add(X, succ(Y), succ(Z)), \neg add(X, Y, Z)\}$

y la cláusula goal

$$\{\neg add(succ(0), V, succ(succ(0)))\}$$

queremos mostrar que el conjunto de estas cláusulas $\{C_1, C_2, G\}$ es insatisfacible.



Y

Cláusula goal	Cláusula de entrada	Sust
$\{\neg add(succ(0), V, succ(succ(0)))\}$	C_2	
$\{\neg add(succ(0), Y, succ(0))\}$	C_1	σ_1
\square		σ_2

donde

$$\sigma_1 = \{X \leftarrow succ(0), V \leftarrow succ(Y), Z \leftarrow succ(0)\}$$

$$\sigma_2 = \{U \leftarrow succ(0), Y \leftarrow 0\}$$

y la sustitución resultado es

$$\sigma_2 \circ \sigma_1 = \{X \leftarrow succ(0), \\ V \leftarrow succ(0), \\ Z \leftarrow succ(0), \\ U \leftarrow succ(0), \\ Y \leftarrow 0\}$$

3.4.5 Corrección y completitud

- **Corrección:** Si un conjunto de cláusulas de Horn tiene una refutación SLD, entonces es insatisfacible.
- **Completitud:** Si un conjunto de cláusulas de Horn $P \cup \{G\}$ como fue descrito, si es insatisfacible, existe una refutación SLD cuya primera cláusula es G.

Si el algoritmo de resolución original era correcto, entonces este también, ya que impone restricciones y se aplica sobre un subconjunto de las fórmulas.

La completitud se preserva solamente para cláusulas de Horn.

3.5 Prolog

Ejemplo. Ejemplo de programa de prolog

```
% Hechos
habla(ale, ruso).
habla(juan, ingles).
habla(maria, ruso).
habla(maria, ingles).

% Reglas de inferencia
seComunicaCon(X, Y) :- habla(X, L), habla(Y, L), X \= Y.

% Ejemplo de goal
seComunicaCon(X, ale)
% X = maria
seComunicaCon(X, Y)
% X = maria, Y = juan y le podemos pedir más
```

Depende de como instancie un goal, puede funcionar distinto

3.5.1 Resolución SLD en Prolog

Prolog usa resolución SLD con las siguientes restricciones:

- **Regla de búsqueda:** se seleccionan las cláusulas del programa de arriba hacia abajo, en el orden que fueron introducidas.
- **Regla de selección:** se selecciona el átomo de más a la izquierda.

La suma de ambas se llama **estrategia**, y cada una determina un árbol de búsqueda o **árbol SLD** distinto.

Sobre la regla de selección, nosotros tenemos un super goal que puede tener muchos literales negados, y selecciona el que está más a la izquierda (nos da determinismo). Esto permite que alteremos el orden el que aparecen las cosas según como queremos que ejecute.

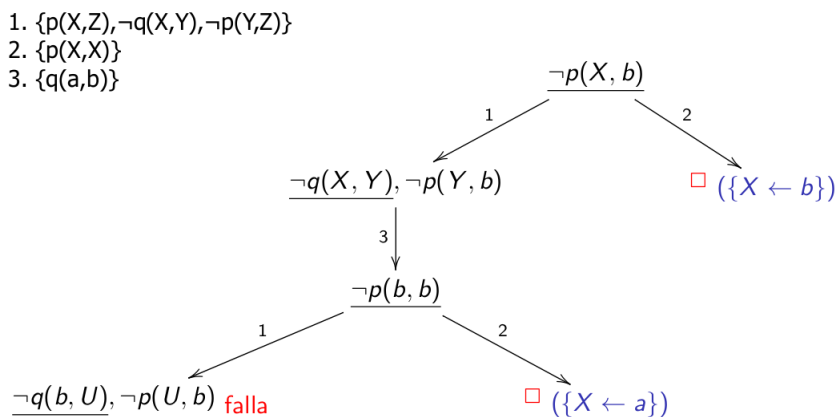
Ejemplo. Ejemplo de resolución SLD en Prolog

Cláusulas de definición:

1. $\{p(X, Z), \neg q(X, Y), \neg p(Y, Z)\}$
2. $\{p(X, X)\}$
3. $\{q(a, b)\}$

Cláusula goal: $\{\neg p(X, b)\}$

El árbol de búsqueda seleccionando el átomo que está más a la izquierda es



1. Camino 1:

- $\sigma_1 = \{Z \leftarrow b\}$
- $\sigma_2 = \{X \leftarrow a, Y \leftarrow b\}$
- $\sigma_3 = \{X \leftarrow b\}$

La sustitución final es $\sigma_3 \circ \sigma_2 \circ \sigma_1 = \{Z \leftarrow b, X \leftarrow a, Y \leftarrow b\}$.

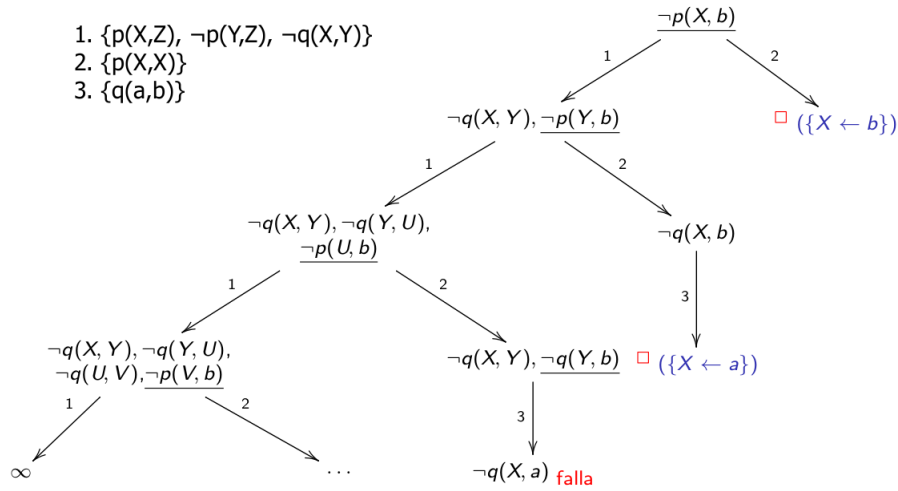
(tiene prioridad el $X \leftarrow a$ por sobre el $X \leftarrow b$ por como se comporta la composición de sustituciones, ver [section 1.4.6 Composición de sustituciones](#))

2. Camino 2: lleva directo a $\sigma_1 = \{X \leftarrow b\}$.

Variando estrategia

Se podría variar la **regla de selección** para que elija el átomo de más a la derecha por ejemplo, lo que llevaría a un árbol SLD distinto.

Ejemplo. Eligiendo el átomo de más a la derecha



nos puede pasar que el árbol diverge, lo cual no quiere decir que no pueda encontrar soluciones.

También se podría variar la regla de búsqueda, en cuyo caso también podría variar el árbol SLD asociado. Por ejemplo, podría seleccionar las cláusulas de abajo hacia arriba en vez de arriba hacia abajo.

La importancia de entender la estrategia que se usa (en Prolog) es que podemos estructurar nuestro programa para tener mejores árboles SLD (y que sea más eficiente).

3.5.2 Motivación de resolución SLD como cómputo

Vamos a explorar de qué manera la **resolución SLD** puede usarse para **computar**. Además vemos el rol de la **sustitución respuesta** como "resultado del cálculo".

Vamos a especificar con fórmulas un cómputo, hacer una pregunta y mediante resolución vamos a obtener una sustitución que es la respuesta.

Recordando el ejemplo de la suma, que puede verse como una definición recursiva de la suma,

- $C_1 = \{add(U, 0, U)\}$
 $(U + 0 = U)$
- $C_2 = \{add(X, succ(Y), succ(Z)), \neg add(X, Y, Z)\}$
 (si $X + Y = Z$, entonces $X + succ(Y) = succ(Z)$)

Y queremos saber si, dada esa definición, "Existe V tal que $1 + V = 2$?"

$$\exists V.add(succ(0), V, succ(succ(0)))$$

Esto podemos plantearlo como la **validez** de la fórmula

$$C_1 \wedge C_2 \supset \exists V.add(succ(0), V, succ(succ(0)))$$

(la definición de la suma implica que existe un valor tal que $1 +$ el da 2)

es decir,

$$\underbrace{C_1 \wedge C_2}_{\text{Define la suma}} \supset \underbrace{\exists V.add(succ(0), V, succ(succ(0)))}_{\text{Pide calcular } V} \text{ es válida}$$

que es lo mismo que preguntarse por la **insatisfactibilidad** de la negación

$$\begin{aligned}
& \neg(C_1 \wedge C_2 \supset \exists V.add(succ(0), V, succ(succ(0)))) \\
& \iff \neg(\neg(C_1 \wedge C_2) \vee \exists V.add(succ(0), V, succ(succ(0)))) \\
& \iff C_1 \wedge C_2 \wedge \neg \exists V.add(succ(0), V, succ(succ(0))) \\
& \iff C_1 \wedge C_2 \wedge \forall V.\neg add(succ(0), V, succ(succ(0)))
\end{aligned}$$

Que son tres cláusulas, y lo podemos escribir como

$$\left\{ \begin{aligned} & \{add(U, 0, U)\}, \\ & \{add(X, succ(Y), succ(Z)), \neg add(X, Y, Z)\}, \\ & \{\neg add(succ(0), V, succ(succ(0)))\} \end{aligned} \right\}$$

y con ese conjunto de cláusulas disparamos la resolución SLD. Si tiene éxito, con la sustitución respuesta obtenemos un valor para el V buscado. Es importante observar que no solo nos interesa saber que exista ese V , queremos una instancia del mismo.

3.5.3 Notación Prolog

La resolución SLD partía de un conjunto de cláusulas $S = P \cup \{G\}$ donde

1. P es un conjunto de cláusulas **definición**, cláusulas con exactamente un literal positivo.

$$\begin{aligned}
& \{B, \neg A_1, \dots, \neg A_n\} \\
& \{B\}
\end{aligned}$$

2. G es un **goal**, una cláusula negativa

$$\{\neg A_1, \dots, \neg A_n\}$$

Notar que

$$\begin{aligned}
B \vee \neg A_1 \vee \dots \vee \neg A_n & \iff \neg(A_1 \wedge \dots \wedge A_n) \vee B \\
& \iff (A_1 \wedge \dots \wedge A_n) \supset B
\end{aligned}$$

como consecuencia, las cláusulas en P definen implicaciones o cosas que son verdaderas y se escriben

- $B : \neg A_1, \dots, A_n$. para $\{B, \neg A_1, \dots, \neg A_n\}$ (**reglas**)
- B . para $\{B\}$ (**hechos**)

3.5.4 Ejemplos

Suma

El ejemplo de la suma en Prolog se escribiría como

```

add(U, 0, U).
add(X, succ(Y), succ(Z)) :- add(X, Y, Z).

% Ingresamos el goal
?- add(succ(0), V, succ(succ(0)))

% La respuesta es
V = succ(0)

```

Familiares

```
hijo(fred, sally).
hijo(tina, sally).
hijo(sally, john).
hijo(sally, diane).
hijo(sam, bill).
hermanos(A, B) :- hijo(P, A), hijo(P, B), A \= B.
```

Si el goal es `hermanos(john, X)`, entonces la refutación SLD para $P \wedge \forall X. \neg \text{hermanos}(\text{john}, X)$ nos va a dar la sustitución respuesta $\sigma = \{X \leftarrow \text{diane}\}$

Puede que la sustitución respuesta asigne valores a variables intermedias que surgieron en el proceso de búsqueda de una refutación.

Secuencia SLD (mía):

1. $\{\text{hijo}(\text{fred}, \text{sally})\}$
2. $\{\text{hijo}(\text{tina}, \text{sally})\}$
3. $\{\text{hijo}(\text{sally}, \text{john})\}$
4. $\{\text{hijo}(\text{sally}, \text{diane})\}$
5. $\{\text{hijo}(\text{sam}, \text{bill})\}$
6. $\{\text{hermanos}(A, B), \neg \text{hijo}(P, A), \neg \text{hijo}(P, B), \neg A \neq B\}$
7. (goal) $\neg \text{hermanos}(\text{john}, X)$
8. (7 y 6) $\{\neg \text{hijo}(P, \text{john}), \neg \text{hijo}(P, B), \neg \text{john} \neq B\}$
 $\sigma = \{A \leftarrow \text{john}, X \leftarrow B\}$
9. (8 y 3) $\{\neg \text{hijo}(\text{sally}, B), \neg \text{john} \neq B\}$
 $\sigma = \{P \leftarrow \text{sally}\}$
10. (9 y 4) $\{\text{john} = \text{diane}\}$
 $\sigma = \{B \leftarrow \text{diane}\}$
11. \square

Acá no estoy seguro de cómo se debería tratar formalmente el $=$, pero supongo que como $\text{john} = \text{diane}$ es falso se puede resolver y queda la cláusula vacía

la sustitución resultado es $\sigma = \{A \leftarrow \text{john}, X \leftarrow \text{diane}, P \leftarrow \text{sally}, B \leftarrow \text{diane}\}$

3.5.5 Corrección y completitud

Corrección: Si existe una refutación SLD de $P \cup \{G\}$ con la estrategia de antes, entonces es insatisfactible.

Completitud: Si $P \cup \{G\}$ es insatisfactible, entonces existe una refutación SLD con la estrategia de antes a partir del él.

3.5.6 Refutaciones SLD en Prolog

Prolog recorre el árbol SLD en **profundidad** (hace DFS). La ventaja es que puede ser implementado de manera muy eficiente,

- Se usa una **pila** para representar los átomos del goal
- Se hace **push** del resolvente del átomo del tope de la pila con la cláusula de definición

- Se hace **pop** cuando el átomo del tope de la pila no unifica con ninguna cláusula de definición más. Luego, el átomo que queda en el tope se unifica con la siguiente cláusula de definición.

Pero tiene como desventaja que puede que no encuentre una resolución SLD aún si existe.

3.5.7 Aspectos extra lógicos

Dos temas de Prolog que trascienden la lógica subyacente:

1. **Cut**: No permiten controlar como se crea el árbol de resolución (podarlo o hacerlo más eficiente)
2. Deducción de información negativa, negation as failure (**negación por falla**): cómo podemos inferir información sobre cosas que *no* pasan

Cut

Es un predicado 0-ario **!** que solo tiene éxito la primera vez que se lo invoca (no puedes pasar de vuelta por ahí, limita las opciones de backtracking). Brinda un mecanismo de control que permite hacer *prunning* del árbol SLD (podar).

Es extra lógico (no se corresponde con un predicado estándar de LPO), y está por cuestiones de eficiencia. Pero hay que tener cuidado, porque puede podarse una rama de éxito deseada.

Ejemplo. Ejemplo de cut

```
p(a).
p(b).
p(c).
q(a, e).
q(a, f).
q(b, f).

% Consultas

?- p(X).
X = a; X = b; X = c;
no

?- p(X), !.
X = a;
no
% p(X) va a resolver a cut, que la primera vez tiene éxito, pero elimina
% las posibilidades de hacer backtracking por lo que no puede resolver p(X)
% para otros X.

?- p(X), q(X, Y).
X = a, Y = e;
X = a, Y = f;
X = b, Y = f;
no

?- p(X), !, q(X, Y).
X = a, Y = e;
X = a, Y = f;
no
% Acá es lo mismo, pero la resolvente es !, q(X, Y), porque lo que se puede
% hacer backtracking qué resolver q(X, Y), pero no sobre qué resolver p(X)
```

Ejemplo. Otro ejemplo

```
p(a).
p(b).
p(c).
q(a, e).
q(a, f).
q(b, f).
r(X, Y) :- p(X), !, q(X, Y).
```

% Consultas

```
?- r(X, Y).
X = a, Y = e;
X = a, Y = f;
no
```

```
?- p(X), r(X, Y).
X = a, Y = e;
X = a, Y = f;
X = b, Y = f;
no
```

% No mata las opciones del p(X) de "arriba".

*% Podas la rama de la cláusula definición donde aparece el cut, pero no
% de más arriba.*

Esto funciona así porque **las ramas que se podan son la de la cláusula de definición donde aparece el cut, pero no de más arriba.**

Por ej. si tenemos $p(X)$, $s(Y)$, $!$, $q(X, Y)$ (suponiendo que $s(e)$, $s(f)$) el $!$ también cortaría esas posibilidades.

```
?- p(X), s(Y), q(X, Y).
X = a, Y = e ;
X = a, Y = f ;
X = b, Y = f ;
false.

?- p(X), s(Y), !, q(X, Y).
X = a, Y = e ;
false.
```

Ejemplo. Definición de max

```
max1(X, Y, Y) :- X <= Y.
max1(X, Y, X) :- X > Y.
```

% Consulta

```
?- max1(3, 4, Z).
Z = 4;
false. % No queremos que pruebe más!
```

es ineficiente, porque puede intentar de buscar más soluciones. Uno puede usar cut para sacarse de encima este problema.

```
max2(X, Y, Y) :- X <= Y, !.
max2(X, Y, X) :- X > Y.
```

```
% Consulta
?- max2(3, 4, Z).
Z = 4.
```

otra opción más,

```
max3(X, Y, Y) :- X =< Y, !.
max3(X, Y, X).
```

```
% Consulta
?- max3(3, 4, Z).
Z = 4.
```

```
% Pero no es válido, porque acá devuelve true de una en vez de ver la otra
?- max2(2, 3, 2).
true. % pero es falso esto
% no puede unificar con la primera, entonces unifica con la 2da de una.
```

En general,

- Cuando se selecciona un cut, tiene éxito inmediatamente
- Si debido al backtracking se vuelve a ese cut, su efecto es hacer fallar el goal que le dio origen. (el goal que unificó con la cabeza de la cláusula que **contiene** al corte y que hizo que esa cláusula se activara)
- El efecto obtenido es el de **descartar soluciones** (no dar más) de
 1. otras cláusulas del goal padre
 2. cualquier goal que ocurre a la izquierda del corte en la cláusula que contiene el corte
 3. todos los objetivos intermedios que se ejecutaron durante la ejecución de los goals precedentes.

Otra forma de verlo, sacada de [CS206 Lecture 16](#).

El **Goal padre** (*parent goal*) es el goal que matcheo con la cabeza de la cláusula que contiene el cut.

Cuando un cut es encontrado, se **resuelve** inmediatamente.

El **efecto** del cut es que el sistema se comprometa (*commit*) a todas las elecciones realizadas desde el momento en el que el goal padre se llamó hasta que se resolvió el cut (incluyendo al goal padre). Es decir, todas alternativas pendientes entre el goal padre y el cut se descartan.

Por ejemplo, si tenemos

```
C :- P, Q, R, !, S, T, U.
C :- V.
A :- B, C, D.
```

Cuando se intenta de resolver A, intentamos C luego de obtener la primera solución para B. Acá se resuelven de a una P, Q, R y se encuentra el cut, que se resuelve inmediatamente. En este momento la resolución se "compromete" a las elecciones realizadas para P, Q y R y no va a backtrackear para buscar otras. Tampoco va a intentar matchear C con otra cabeza, por ejemplo C :- V. Pero si va a devolver todas las nuevas respuestas obtenidas de S, T y U.

El cut es invisible a A, y todas las opciones de B y D se van a probar.

Not - Negación por falla

En cláusulas de Horn no hay una forma directa de escribir negaciones. Si uno las traduce, significan que un hecho es true.

Dado un árbol SLD, podemos tener ramas de falla. Si tenemos un árbol que falla **finitamente** (tiene ramas de falla finitas y no tiene ramas de éxito)

Dado un programa P el **conjunto de falla finita** de P es

$\{B \mid B \text{ es átomo cerrado ('ground')} \text{ y existe un árbol SLD que falla finitamente con } B \text{ como raíz}\}$

Un átomo cerrado es uno que no tiene variables, por ej. $P(a)$ a dif de $P(x)$.

Infiere la regla de negation as failure

$$\frac{B \text{ átomo cerrado} \quad B \text{ en conjunto de falla finita de } P}{\neg B}$$

Esto en prolog se implementa con el **predicado not**,

```
not(G) :- G, !, fail.  
not(G).
```

not(G) matchea con el primero. Llamá a G. Si encontraste algo en tu base de conocimiento que lo resuelve, cortá (cut) y fallá. Pero si no encontraste nada, el primero falla, y la 2da definición da true.

En otras palabras, si tenés éxito fallás y si fallás tenés éxito.

Es importante que not(G) contenga cut (!). De no ser así, siempre devolvería true, ya que aunque falle para la primera luego resolvería por la segunda.

Se puede probar implementando

```
not2(G) :- G, fail
```

Ejemplo. Podemos deducir not(student(mary)) y not(student(anna)) a partir de

```
student(joe).  
student(bill).  
student(jim).  
teacher(mary).
```

Pero negación por falla **no es la negación lógica**

Ejemplo. Por ejemplo

```
animal(perro).  
animal(gato).  
vegetal(X) :- not(animal(X)).
```

La consulta vegetal(perro) da no como se espera, idem vegetal(pasto) que da si.

Pero vegetal(X) da falso. (porque no es un átomo cerrado).

Para que estos predicados funcionen, es necesario que esté instanciado lo que quiero negar. El goal not(G) **nunca instancia variables** de G.

- Si G tiene éxito, fail falla y descarta la sustitución
- Caso contrario, not(G) tiene éxito inmediatamente (sin afectar ni contrar sustituciones para G).

por lo tanto, not(not(animal(X))) no es equivalente a animal(X), da true.

Porque

```
not(not(animal(X)))  
(G = not(animal(X)))  
not(animal(X)), !, fail
```

y como `not(animal(X))` da `false`, el G de arriba falló, y `not` tenía éxito cuando G fallaba, por lo que el `not` de arriba tiene éxito y el resultado total es `true`.

Ejemplo. Otro ejemplo

```
firefighter_candidate(X) :-
    not(pyromaniac(X)),
    puntual(X).
pyromaniac(attila).
puntual(jeanne_d_arc).

firefighter_candidate2(X) :-
    puntual(X),
    not(pyromaniac(X)).

?- firefighter_candidate(W).
false.

?- firefighter_candidate2(W).
W = jeanne_d_arc.
```

El resultado de `firefighter_candidate(W)` es `false`, porque estoy llamando al `not` sin el `X` instanciado. En cambio, en `firefighter_candidate2(W)` como primero instancia el `X` con `puntual`, el `not` se llama con `X` instanciado y funciona como es esperado